BASIC Reference

FLUKE®

Digitized by the Internet Archive
in 2025

https://archive.org/details/manualsplus_09912

BASIC Reference

FLUKE®

BASIC Reference

P/N 718833
January 1984
©1984 John Fluke Mfg. Co., Inc.
All rights reserved. Litho in U.S.A.

FLUKE®

BASIC

Quick Reference Card

FLUKE®

NOTE

The **REF** column refers to the Reference number used in the Reference Section for the Basic Keyword.
The quantities X, Y, n%, A$, B$, etc. used in the following examples are the arguments received by the statement, function or command.
I.M.C. = Immediate Mode Command

REF	KEYWORD	ACTION
1	ABS(X) Function	Returns |X| or the absolute value of its argument.
69	AND Operator	Logical (Boolean) AND operation.
2	Arithmetic Operators	Unary operators, addition, subtraction, multiplication, division, and exponentiation.
3	Array Variables	Describes the format and usage of array variables.
4	ASCII(A$) Function	Returns the decimal position of the ASCII character.
5	ASH(X%) Function	Signed arithmetic shift of integer.
6	ASSIGN Function	Assign a device as system device (SY0:).
7	ASSIGNMENT Operator	Using "=" to assign values.
8	ATN(X) Function	Returns arctan(X) in radians.
9	BREAK Statement	Send break signal on RS-232 port.
10	CALL Statement	Execute a FORTRAN or Assembly Language program.
11	CHR$(X%) Function	Yields a character equal to the ASCII equivalent of X.
12	CLEAR Statement	Send IEEE-488 device clear signal to device or devices.
13	CLOSE Statement	Closes a channel previously opened to a port, device, or file.
14	CMDFILE Function	Logical function: is a command file active?
15	CMDLINE$ System Function	Read the current FDOS command line.
16	COM Statement	Reserve variables and arrays in common area for chained programs.
17	CONFIG Statement	Configure or unconfigure instrument for parallel poll.
18	CONT TO I.M.C.	Resume program execution at linenumber from breakpoint caused by STOP, STOP ON, CONT TO, or CTRL /C.
19	COPY Statement	Copy a file.
20	COS(X) Function	Returns cosine(X) in radians.
21	CPOS(X%,Y%) Function	Returns the string needed to position the cursor at the given X,Y coordinates.
117	DATA Statement	Define sequence of data items to be read by READ.
150	DATE$ System Variable	Returns date as DD-MM-YY.
22	DEF FN Statement	Define function.
23	DELETE I.M.C.	Delete program lines in memory.
24	DIM Statement	Dimension array or file.

Form No. F740 Rev 1 1/84

©1984 John Fluke Mfg. Co., Inc.,
all rights reserved. Litho in U.S.A.

REF	KEYWORD	ACTION
25	DIR Statement	Print directory of file structured device (like FUP /L).
26	DISABLE Statement	Disable all interrupts (except ON ERROR).
27	DISABLE CMDFILE Stmt	Suspend execution of command file.
28	DUPL$(A$) Function	Create a string of duplicated characters.
29	EDIR Statement	Print extd. directory of file structured device (like FUP /E).
30	EDIT I.M.C.	Enter the EDIT mode.
31	ENABLE Statement	Enable interrupts after a DISABLE Statement.
32	ENABLE CMDFILE Stmt	Resume command file execution (see DISABLE CMDFILE).
33	END Statement	The last program statement, causes exit to OK prompt.
150	ERL System Variable	Returns line number of last error.
150	ERR System Variable	Returns error number of last error.
34	EXEC Statement	Chain to machine language (.FD2) or command file (.CMD).
35	EXIT	Exit basic, return to FDOS or SHELL.
36	EXP(X) Function	Returns natural log base e raised to power X.
37	FLEN Variable	Returns the length of a file.
38	Floating-Point Variables	Describes floating-point (real) variables, constants, and data.
39	FOR and NEXT Statement	Sets up a program loop.
40	GOSUB Statement	Subroutine call — goto linenumber, do it, come back.
41	GOTO Statement	Branch to linenumber, don't return.
42	IF-GOTO Statement	IF condition GOTO linenumber.
42	IF-THEN Statement	IF condition THEN do this.
42	IF-THEN-ELSE Statement	IF condition THEN do this ELSE do this.
43	INCHAR(X%) Function	Read single character from open channel or console.
44	INCOUNT(X%) Function	Number of characters/lines available from serial device.
45	INIT Statement	Initialize IEEE-488 port.
46	INPUT Statement	Get data from user.
47	INPUT #n Statement	Data input from system level device through channel.
48	INPUT Statement	IEEE-488 Bus input.
49	INPUT LINE Statement	Get string data from user.
50	INPUT LINE #n Statement	Get string data from I/O channel.
51	INPUT LINE @ Statement	Get string data from IEEE-488 Bus.
52	INPUT LINE WBYTE Stmt	IEEE-488 Bus input with prior Bus message.
53	INPUT WBYTE Statement	Send IEEE-488 Bus message prior to getting data item.
54	INSTR(n%,A$,B$) Function	Returns the first character position of a substring B$ within string A$ starting at n%.
55	INT(X) Function	Return the greatest integer ⩽ X.
56	INTEGER Variables	Describes the format and usage of integer variables, constants, and data.
57	KEY System Variable	Returns touch sense "key" last pressed.
58	KILL Statement	Remove, erase this file from device.
59	LCASE$(A$) Function	Convert string to lower-case.
60	LEFT(A$,n%) Function	Return the leftmost n% characters of A$.
61	LEN(A$) Function	Return the number of characters in A$.
62	LET Statement	Assignment operator — LET X = 5.
63	LINK Statement	Load an object file into system RAM.
64	LIST I.M.C.	LIST program lines.
65	LN(X) Function	Return the natural log (log e) of X.
66	LOCAL Statement	Reset IEEE-488 Bus instruments to local state.
67	LOCKOUT Statement	Implement local lockout on the IEEE-488 Bus.
68	LOG(X) Function	Return the log (log 10) of X.
69	Logical Operators	Describes the AND, OR, XOR, and NOT operators.
70	LSH(X) Function	Unsigned logical integer shift (see ASH).
71	MATH Functions	Describes the mathematical functions.
72	MEM System Variable	Returns amount of available memory in bytes.
73	MID(A$,s%,n%) Function	Return substring of A$ given starting position s% and number of characters n%.
74	MOD(X) Function	Return remainder of division of division operation.
39	NEXT Statement	Closes FOR-NEXT loop.
69	NOT Operator	Logical (Boolean) NOT operation.
75	NUM$(X) Function	Convert X to a string as PRINT or PRINT USING would output it.

REF	KEYWORD	ACTION
76	OFF CTRL/C Statement	Disable CTRL /C interrupt processing.
77	OFF #n Statement	Disable interrupt processing from channel #n.
78	OFF CLOCK Statement	Disable clock interrupts.
79	OFF ERROR Statement	Disable action of previous ON ERROR GOTO statement.
80	OFF INTERVAL Statement	Disable interval interrupts.
81	OFF KEY Statement	Disable interrupt from Touch Sensitive Overlay.
82	OFF PORT Statement	Disable interrupt sensing on port.
83	OFF PPOL Statement	Disable parallel poll interrupt processing.
84	OFF PPORT Statement	Disable PPORT status checking.
85	OFF SRQ Statement	Disable service request interrupt processing.
86	OLD Statement	Load a program into memory from mass storage, do not run.
87	ON CLOCK Statement	Create interrupt at specific time of day.
88	ON CTRL/C Statement	Enable CTRL /C interrupt processing.
89	ON ERROR Statement	Enable interrupt processing on error condition.
90	ON #n Statement	Enable interrupt processing from channel #n.
91	ON...GOSUB Statement	ON condition GOSUB this linenumber.
91	ON...GOTO Statement	ON condition GOTO this linenumber.
92	ON INTERVAL Statement	Create interrupt at interval (SET INTERVAL).
93	ON KEY Statement	Enable interrupt from Touch Sensitive Overlay.
94	ON PORT Statement	Enable interrupt sensing on port when not controller-in-charge.
95	ON PPOL Statement	Enable parallel poll interrupt processing.
96	ON PPORT Statement	GOTO from parallel port interrupt.
97	ON SRQ Statement	Enable service request interrupt processing.
98	OPEN Statement	OPEN a channel to a file, device, or port.
99	OPEN Statement	OPEN a channel to a virtual array.
69	OR Operator	Logical (Boolean) OR operation.
100	PACK Statement	Pack a file structured device.
101	PASSCONTROL Statement	Give up IEEE-488 control to another controller.
102	PI User Constant	$\pi = 3.14159265358979$.
102a	PORT Expression	Direct address to port
103	PORTSTATUS(X%) Function	Determine status of interface port.
104	PPL Function	Parallel poll Bus port and return result.
105	PRINT Statement	PRINT something (usually on the controller's display).
106	PRINT @ Statement	Output something to the IEEE-488 Bus.
107	PRINT #n Statement	PRINT something to a channel.
108	PRINT USING Statement	PRINT something USING this format.
109	PROTECT Statement	Set delete protection for a file.
110	QDIR Statement	Quick directory listing of file-structured device.
111	RAD(x%,b%) Function	Integer x% to non-decimal base b% conversion.
112	RANDOMIZE Statement	Shake up the random number generator.
113	RBIN Statement	Receive binary data from IEEE-488 instruments.
114	RBIN WBYTE Statement	Output data specified by WBYTE, then perform RBIN.
115	RBYTE Statement	Read fixed-length byte from IEEE-488 instruments.
116	RBYTE WBYTE Statement	Output data specified by WBYTE, then perform RBYTE.
117	READ, DATA, and RESTORE Statement	READ the data from the DATA statements, RESTORE (resume) reading of data from the DATA statement.
118	Relational Operators	Describes =, >, <, <>, <=, and >= operators.
119	REM Statement	REMark, comment. Comment your programs.
120	REMOTE Statement	Set IEEE-488 Remote Enable line to true.
121	REN Immediate Mode Cmd	Renumber the program in memory.
122	RENAME Statement	Rename a file.
124	RESAVE Statement	Non-interactive SAVE.
124	RESAVEL Statement	Non-interactive SAVEL.
117	RESTORE Statement	Resume or re-read DATA statements(s).
125	RESUME Statement	Resume program operation after interrupt.
126	RETURN Statement	Subroutine return: go back to calling program after GOSUB.
127	RIGHT(A$,n%) Function	Return a substring of A$ starting at n% to the end of A$.
128	RND Function	Return a pseudorandom number.
129	RUN I.M.C.	Run a BASIC program.
129	RUN Statement	Chain to another BASIC program.
130	SAVE Command	SAVE program in file on device, ask before clobbering old file.
130	SAVEL Statement	SAVE program in lexical form, ask before clobbering old file.
131	SET CLOCK Statement	Set clock interrupt (ON CLOCK...GOTO).
132	SET CMDLINE$ Statement	Set the FDOS command line.
133	SET DATE Statement	Set the date in the system clock.

REF	KEYWORD	ACTION
134	SET ECHO Statement	Puts the keyboard into normal ("line") mode.
135	SET INTERVAL Statement	Set interval interrupt (ON INTERVAL...GOTO).
136	SET NOECHO Statement	Puts the keyboard into editing ("character") mode.
137	SET SHELL Statement	Use this program when exit occurs, rather than FDOS.
138	SET SRQ Statement	Service request from controller-in-charge.
139	SET TIME Statement	Set the system clock.
140	SGN(X) Function	Return the sign of X (-1, 0, 1).
141	SIN(X) Function	Return sine(X) in radians.
142	SPACE$(n%) Function	Return a string of n% space characters.
143	SPL(n%) Function	Return status byte of instrument by serial poll.
144	SQR(X) Function	Return the square root of X.
144a	STEP I.M.C.	Single Step program execution
145	STIME$ System Variable	Returns the current time in 24 hour format (H:MM:SS).
146	String Comparisons	Describes the usage of relational operators on strings.
147	Strings	Describes string constants, variables, and data.
148	STOP Statement	STOP program execution.
149	STOP ON Statement	STOP program execution on this line number.
150	System Variables	Describes the various system variables.
151	TAB(n%) Function	Move the cursor or printhead to the next tab position.
152	TAN(X) Function	Returns tangent(X) in radians.
153	TERM Statement	Specify terminating character for input data (IEEE-488).
150	TIME System Variable	Returns time in milliseconds since midnight.
150	TIME$ System Variable	Returns time as HH:MM in 24 hour format.
154	TIMEOUT Statement	Limit wait time for response to IEEE-488 Bus request.
154a	TRACE OFF Statement	Disable trace facility
155	TRACE Statement	Line number trace facility.
156	TRIG Functions	Describes trigonometric functions.
157	TRIG Statement	Address specified devices as listeners (IEEE-488).
158	TRIM Statement	Trim trailing null characters from virtual array strings.
159	UCASE$(A$) Function	Convert string to upper-case.
160	UNLINK Statement	Forget previously LINKed .obj file
161	UNPROTECT Statement	Remove delete protection from a file.
162	VAL(A$) Function	Convert numeric string A$ to floating-point.
163	WAIT Statement	Suspend program execution for specified time period.
164	WAIT Statement	Suspend program execution until interrupt occurs.
164	WAIT FOR TIME Statement	Uninterruptible WAIT statement.
165	WAIT FOR KEY Statement	Suspend program execution for TSO input.
166	WAIT FOR PPOL	Suspend program execution for parallel poll on IEEE bus.
167	WAIT FOR SRQ	Suspend program execution for service request on IEEE bus.
168	WBIN Statement	Output single or double precision data (IEEE-488).
169	WBYTE Statement	Send integer array as Bus message to designated port.
69	XOR Operator	Logical (Boolean) exclusive OR operation.

REF	SYMBOL	ACTION
56	n%	Integer variable or data.
147	A$	String variable or data.
3	A$(X,Y)	String array (2 dimensional).
3	A%(X,Y)	Integer array (2 dimensional).
3	A(X,Y)	Floating-point variable array.
7	=	Assignment operator (let X = Y).
118	=	Relational equality operator.
2	+	Addition operator (X = Y + 3).
146	+	String concatenation operator.
2	-	Subtraction operator (X = Y–3).
2	*	Multiplication operator (X = Y * 5).
2	/	Division operator (X = Y / 2).
2	^	Exponentiate (X^2 = X * X).
118	<	Relational less than operator.
118	<=	Relational less than or equal to operator.
118	>	Relational greater than operator.
118	>=	Relational greater than or equal to operator.
118	<>	Relational not equal operator.
119	!	Denotes comment following "!" symbol.

ERROR CODES

TYPE	NUMBER	LEVEL	ERROR
OVERFLOW	0	F	User storage overflow
	1	F	Virtual array declaration too large (more than 64K-bytes)
	2	F	File too small for virtual array declarations
SYSTEM	100	F	BASIC or FDOS system error
	101	F	Configuration error -- incompatible ".BAL" file format
COMMAND	200	F	Immediate mode command in RUN mode or vice-versa
	201	F	Cannot CONTinue at this point
	202	F	STEP outside break mode is not permitted
I/O	300	R	Diskette is not loaded
	301	R	Diskette is write protected
	302	R	Illegal channel number (<1 or >16)
	303	R	Channel already in use
	304	R	Invalid device name
	305	R	File not found on device
	306	R	No room left on device
	307	R	Read/write past physical end of file
	308	R	Channel is not open
	309	R	Data lost on RS-232 channel input
	310	W	Input line too long
	311	R	Non-recoverable device read/write error
	312	R	Illegal file name specified
	313	F	Channel is not open as random (DIM) file
	314	F	Sequential access to random (DIM) file
	315	F	Virtual array assigned to non-file-structured device
	316	F	Illegal lexical (.BAL) file name
	317	R	Invalid device directory
	318	R	Read from output file or write to input file not permitted
	319	R	Wrong device type for ON n interrupts
	320	F	Illegal object file format
	321	R	Device directory is full
	322	R	Illegal operation for device type
	323	R	File is delete protected – can't KILL or OPEN AS NEW FILE
	324	R	Can't RENAME file
	325	R	File medium (diskette) was swapped
	326	R	Can't load program - too little memory is available
	327	R	Illegal image (.FD2) file format
	328	R	Command line too long (maximum 80 characters)
	329	R	RS-232 port number outside range (> 9) for BREAK
	330	R	Parallel port number out of range (>15)
INSTRUMENT BUS CONTROL	400	R	Illegal IEEE-488 port number specified (<0 or >1)
	401	R	Illegal IEEE-488 device primary address(<0 or >30)
	402	R	Illegal IEEE-488 device secondary address (<0 or >31)
	403	R	IEEE-488 bus handshake incomplete
	404	R	Too many IEEE-488 ports specified for function
	405	R	No devices attached to IEEE-488 port
	406	R	No IEEE-488 ports available
	407	R	Unavailable IEEE-488 port specified
	408	R	Timeout during IEEE-488 bus I/O
	409	R	Illegal WBYTE data (both EOI and ATN specified)
	410	R	Parallel poll bit number out of range (<1 or >8)
	411	R	Parallel poll bit sense out of range (<0 or >1)
	412	R	IEEE-488 bus timeout value out of range (<0 or >32767)
	413	R	IEEE-488 TERM string longer than 1 character
	414	R	No IEEE-488 driver configured in FDOS
	415	W	Illegal SET SRQ status byte value (<0 or >255)
	416	R	Illegal operation for current IEEE-488 port state
SYNTAX	500	F	Unrecognized statement
	501	F	Illegal character terminating statement
	502	F	Badly formed array subscript
	503	F	Mismatched parentheses in expression
	504	F	Illegal LET statement
	505	F	Illegal IF statement
	506	F	Illegal line number
	507	F	Illegal PRINT statement
	508	F	Illegal PRINT USING mask
	509	F	Illegal INPUT statement
	510	F	Illegal DIMension statement
	511	F	Badly formed DEFine function statement
	512	F	Illegal FOR statement

ERROR CODES (cont.)

TYPE	NUMBER	LEVEL	ERROR
SYNTAX (cont)	513	F	FOR without NEXT
	514	F	NEXT without FOR
	515	F	Mismatched quotes in string constant
	516	F	Ill-formed expression
	517	F	Illegal OPEN statement
	518	F	Illegal CLOSE statement
	519	F	IEEE-488 statement syntax error
	520	F	Illegal COMmon statement
	521	F	RETURN statement encountered when NEXT statement required
	522	F	Illegal variable name
	523	F	ON statement syntax error
	524	F	OFF statement syntax error
	525	F	TRACE statement syntax error
	526	F	Illegal file SIZE specification
	527	F	RENumber parameter error
	528	F	RENumber statement syntax error
	529	F	ELSE without IF
	530	F	NEXT syntax error
	531	F	INPUT WBYTE requires IEEE-488 input
	532	F	Illegal array subrange specified
	533	F	WBYTE/RBYTE data not integer type
	534	F	Can't specifiy column for WBYTE/RBYTE array
	535	F	Can't use undimensioned variable for WBYTE/RBYTE
	536	F	Can't use virtual arrays with WBYTE/RBYTE
	537	F	2-dimensional array illegal with WBYTE/RBYTE
	538	F	Illegal CONFIG statement
	539	F	Illegal RBYTE statement syntax
	540	F	RBYTE cycle length must be greater than zero
	541	F	Total number of RBYTE elements not a multiple of cycle length
	542	F	Illegal WBYTE clause syntax
	543	F	Illegal data type for RBIN/WBIN binary transfer
	544	F	WAIT statement syntax error
	545	F	Illegal CALL statement
	546	F	Virtual array parameter illegal for CALL
	547	F	CALL statement parameter syntax error
	548	F	Illegal SET statement syntax or option
	549	F	Require filename for SAVE/SAVEL/RESAVE/RESAVEL
	550	F	Illegal RENAME statement syntax
MATH	600	F	Illegal data type mixing in expression
	601	R	Arithmetic overflow
	602	R	Arithmetic underflow
	603	R	Divide by zero
	604	R	Square root of negative number
	605	R	Exponent too large (max is 708) for exp() function
	606	R	LOG or LN argument <= 0
	607	R	Trigonometric function argument too large (32767 maximum)
	608	R	Improper argument(s) in power
TRANSFER	700	F	Illegal GOTO or GOSUB
	701	F	RETURN without GOSUB
	702	F	Illegal RESUME -- no interrupt active
	703	F	Call to undefined function (no DEF statement for function)
	704	R	ON .. GOTO selector is out of range
	705	F	CALL to undefined subroutine
	706	F	Parameter count mismatch in CALL
	707	R	Illegal time or date specified in SET CLOCK/DATE/INTERVAL/TIME
	708	R	Timer not SET for ON CLOCK/INTERVAL
INPUT	800	R	Out of DATA in READ
	801	W	Too much data typed for INPUT statement
	802	W	Not enough data typed for INPUT statement
	803	W	Illegal character in data for INPUT or VAL function
	804	F	Badly formatted DATA statement
VARIABLE	900	F	Use of non-existent variable value
	901	F	DIM of previously declared variable
	902	R	Array subscript out of range
	903	F	COM of previously declared variable
	904	W	String too long for virtual array string field
	905	F	Incompatible COM declaration -- COM area sizes don't match
	906	F	COM after LINK statement not allowed
	907	F	Illegal XOP 1 function code

F = Fatal R = Recoverable W = Warning

Function

Format

ABS(numeric expression)

Description

The ABS() function returns the absolute value of a floating point or integer number.

- ABS has a floating-point number or integer, as an argument, and returns a positive floating-point number or integer.
- ABS changes a negative value to positive, but has no effect on positive values or zero.
- The domain of input values is any positive or negative floating point or integer value, or zero.
- The range of output values is positive floating point or integer values, and zero.

Examples

The following examples illustrate the usage of the ABS() function. In these examples, x% = –12344, y% = 12345, x = –54 e 99

```
Ready
print abs(x%),abs(y%),abs(x)
 12344            12345            0.54E+101

Ready
_
```

ARITHMETIC Operators 2

+-*/

Description

Arithmetic operators act upon or between numbers or numeric expressions to produce a numeric result. A numeric expression is composed of an arithmetic operator and one or more operands. The following guidelines apply:

- The + and – operators can act upon a single number or numeric expression (unary operation).
- All arithmetic operators act between two numbers or numeric expressions.
- Numeric variables can be used as numbers in expressions if they have previously been assigned a value.
- Floating-point numbers and integers may be intermixed. Integers will automatically be converted to floating-point, if necessary.
- When one integer is divided by another, the result is a truncated integer quotient (the fraction or remainder is truncated). For example:

  2% / 5% is 0%
  15% / 3% is 5%
  17% / (–3%) is –5%.

- When a result is assigned to a numeric variable, it is automatically converted to the assigned data type.
- When a floating-point number is assigned to an integer variable it is rounded, not truncated.
- Computation speed is significantly faster when floating-point and integer data types are not intermixed.
- The result of evaluating an arithmetic expression may be used in a larger expression or assigned to a variable for later use.
- Except for +, arithmetic operators cannot be used on strings.

ARITHMETIC Operators

+-*/

Arithmetic Operators

OPERATOR	NAME	MEANING AND EXAMPLES
+	Positive	Unary plus operator. Does not change sign. + 58.4 + D I% = + KR%
+	Add	Add two numeric quantities. .382 + .046 A + B RE% = CK + T% IF C% = K% + 1% THEN RESUME
-	Negative	Unary minus operator. Changes the sign. - 73.138 - KV C% = - ST%
-	Subtract	Subtract two numeric quantities. 46332.33 - 473.88 C - D T% = RE% - CK IF D% = T% - 10% THEN WAIT FOR KEY
*	Multiply	Multiply two numeric quantities. 44.21 * 3.992 3.582 * RR K% = 4 * I% IF X > 4.77 * CK THEN Y = Y * 5.76
/	Divide	Divide first numeric quantity (dividend) by the second numeric quantity (divisor) to produce a quotient. 3.584 / 0.338 KV / 2% MA = V / KO PRINT C / PI

ARITHMETIC Operators

+-*/

^ Exponentiate — Raise the first numeric quantity to a power equal to the second numeric quantity.

```
PI * R^2%
LS = VR^2.886E-6
```

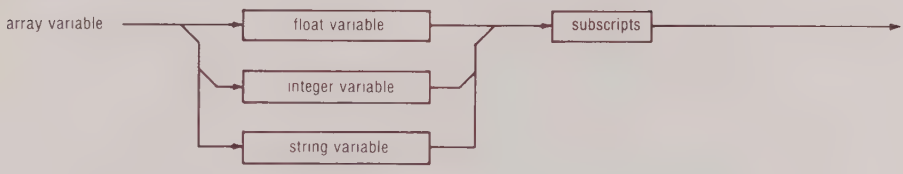
NOTE

Exponentiation is left associative, meaning that ABC is evaluated as (A^B)^C.

Format

legal variable name (row%, column%)

Syntax Diagram



An array variable is a collection of variable data under one name.

- Arrays consist of floating-point, integer, or string variables.
- The variable name has either one or two subscripts to identify individual items within the array.
- Subscripts are enclosed in parentheses.
- When two subscripts are used, they are separated by a comma.
- It is helpful to view two-dimensional arrays as a matrix. The first subscript is the row number, and the second subscript is the column number. For example, FT%(3,18) identifies the integer in row 3, column 18 of the array FT%(m,n).
- A subrange (portion) of an array can be designated by specifying a first and last subscript separated by two periods.

Example

For Example:

A$(3..7)	Strings 3 through 7 of the string array A$.
FT%(2..4, 5..15)	Rows 2 through 4 in columns 5 through 15 of the integer array FT%.

- In the last example above, the second subscript is incremented or decremented before the first. For example, the statement PRINT FT%(2..4,5..15) will display the range FT%(2, 5 through 15) before displaying FT%(3, 5 through 15).

ARRAY Variable

- Array variables are distinct from simple variables. A and A(0) are two different variables.

- Only one array variable can be associated with an identifier. A%(n) and A%(m,n) are not simultaneously allowed.

- Memory space must be reserved for an array variable before it can be used. See the discussion of the DIM statement.

- Virtual arrays are array variables accessible through a channel to a file-structured storage device. This feature allows a program to take advantage of the much greater storage space available on these mass storage devices. Refer to the section on Data Storage.

- Some examples of array variables are:

```
A%(3)
B1%(2%, 3%)
A$(5)
C(3%)
D(2 + A * B, C)
D( D(2) )
```

Format

ASCII(string$)

Description

The ASCII function returns an integer equal to the ASCII (decimal) value of the first character of its string argument.

- ASCII has a string as an argument and yields an integer result.
- The string may be any length, subject only to string length limits.
- Refer to Appendix G, ASCII/IEEE-488 Bus Codes, for a chart of ASCII characters and corresponding decimal numbers.
- Range of possible results is 0 to 255.
- ASCII characters generated by the Instrument Controller's Console give results from 0 to 127. The most significant bit (7) is reset to 0.
- Result is 128 higher when the most significant bit (7) in the character is set to 1. This allows string characters to be examined as 8-bit binary data bytes.
- A null (no characters) string input produces a 0 result.
- The ASCII function may be used in any expression or statement that allows the use of an integer variable.

ASCII()
Function

Examples

The following examples illustrate the results of different uses of the ASCII function. A$ = "NOW" and B$ = "" (null string).

STATEMENT	RESULT
`PRINT ASCII (A$)`	Display 78, the ASCII value of "N", the first character of string A$.
`B% = ASCII (A$)`	Places 78, the ASCII value of "N", the first character of string A$, into integer variable B%.
`IF ASCII (C$) = 13% THEN 1200`	Branch to line 1200 if the first character of string C$ is a Carriage Return.
`PRINT ASCII (B$ + A$)`	Display 78. The null string has an ASCII value of zero: B$ (null) + A$ (NOW) equals "NOW" ("N" is still the first character).
`PRINT ASCII ("N")`	Display 78. A string variable may also be used.

Format

ASH(arg%,count%)

Description

The ASH() function performs a signed arithmetic shift on binary integers (arg%) by shifting it by a given number of bits (count%).

- The function arguments are considered to be integers; any floating-point arguments will be truncated to integer.
- This operation may cause an arithmetic overflow (error 601) to be reported if the argument cannot be represented as an integer.
- The ASH() function shifts arg% right or left by count% bits. This is an arithmetic shift, which means that the sign bit (most significant bit) is propagated (shifted in from the left) if a right shift is performed.

Example

Right Shifts

A right shift is accomplished by using one of the two shift functions, ASH() or LSH(), with a negative shift count. The shift is performed by taking the absolute value of the shift count and then shifting the argument right by that number of bits.

The ASH() (arithmetic shift) function copies the sign bit of a number when performing a right shift. As an example:

```
100 PRINT "Shift count", "Hex", "Decimal", "Binary"
110 FOR I%=0% TO -15% STEP -1%
120   J% = ASH(-32768%, I%)
130   PRINT I%, RAD$(J%, 16%), J%, RAD$(J%,2%)
140 NEXT I%
```

prints the following:

```
Shift Count     Hex     Decimal   Binary
 0              8000    -32768    1000000000000000
-1              C000    -16384    1100000000000000
-2              E000    -8192     1110000000000000
-3              F000    -4096     1111000000000000
-4              F800    -2048     1111100000000000
-5              FC00    -1024     1111110000000000
-6              FE00    -512      1111111000000000
-7              FF00    -256      1111111100000000
-8              FF80    -128      1111111110000000
-9              FFC0    -64       1111111111000000
-10             FFE0    -32       1111111111100000
-11             FFF0    -16       1111111111110000
-12             FFF8    -8        1111111111111000
-13             FFFC    -4        1111111111111100
-14             FFFE    -2        1111111111111110
-15             FFFF    -1        1111111111111111
```

NOTE

The sign bit is copied as the number is shifted to the right, so that when a negative number is right-shifted it remains a negative number.

Left Shifts

A left shift is accomplished by using one of the two shift functions, ASH() or LSH(), with a positive shift count. The shift is performed by shifting the argument left by the number of bits indicated.

In the case of a left shift both ASH() and LSH() generate identical results. The program:

```
100 PRINT "Shift count",, "Hex", "Decimal", "Binary"
110 FOR I%=0% TO 15%
120   J% = ASH(1%, I%)
130   H$ = RAD$(J%, 16%) \ B$ = RAD$(J%,2%)
140   BB$ = DUPL$("0",16-LEN(B$)) + B$
150   PRINT I%, DUPL$("0", 4% - LEN(H$)); H$, J%, BB$
160 NEXT I%
```

prints the following:

Shift Count	Hex	Decimal	Binary
0	0001	1	0000000000000001
1	0002	2	0000000000000010
2	0004	4	0000000000000100
3	0008	8	0000000000001000
4	0010	16	0000000000010000
5	0020	32	0000000000100000
6	0040	64	0000000001000000
7	0080	128	0000000010000000
8	0100	256	0000000100000000
9	0200	512	0000001000000000
10	0400	1024	0000010000000000
11	0800	2048	0000100000000000
12	1000	4096	0001000000000000
13	2000	8192	0010000000000000
14	4000	16384	0100000000000000
15	8000	-32768	1000000000000000

Remarks

Compare the ASH() function to the LSH() function.

ASSIGN 6
Statement

Usage

variable = {expression, variable, or function}

Syntax Diagram

Description

When used as a program statement, with a variable to its left, and an expression to its right, = assigns the result of evaluating the expression to the variable. No equality is implied. This is the default form of the LET statement.

Example

In the following example, the integer N% is incremented by 1.

```
N% = N% + 1%
```

This example shows a variable assigned tothe value of a function:

```
T$=STIME$
```

ASSIGNMENT 7
Operator

Format

variable = expression
variable = variable
variable = function

Description

When used as a program statement, with a variable to its left, and an expression to its right, = assigns the result of evaluating the expression to the variable. No equality is implied. This is the default form of the LET statement.

Example

In the following example, the integer N% is incremented by 1.

```
N% = N% + 1%
```

This example shows a variable assigned to the value of a function:

```
T$=STIME$
```

ATN() Function

Format

ATN(numeric expression)

Description

The ATN function returns the principal arctangent, in radians, of a floating-point numeric input value.

ATN has a floating-point number as an argument, and returns a floating-point result.

The arctangent is the inverse of the tangent. ATN returns an angle, expressed in radians, whose tangent value is the given input.

The range of input values is any floating-point number. The range of output values is between and including the limits of $\pm$ PI/2. Input values within approximately 1E-16 of 0, result in an output of 0. Underflow error does not occur.

Example

```
Ready
PRINT ATN(45)
 1.548578

Ready
```

BREAK Statement

Usage

BREAK device%

Syntax Diagram

Description

The BREAK statement sends a break signal to the RS-232 port designated by device%. Device% is the device number of a RS-232 port and is derived from the device name as follows.

device%	device name
1	kb1:
2	kb2:
3	kb3:
...	...
9	kb9:

Error 329 (illegal break parameter) is reported if the value of device% is outside of the range: 1 <= device% <= 9.

Example

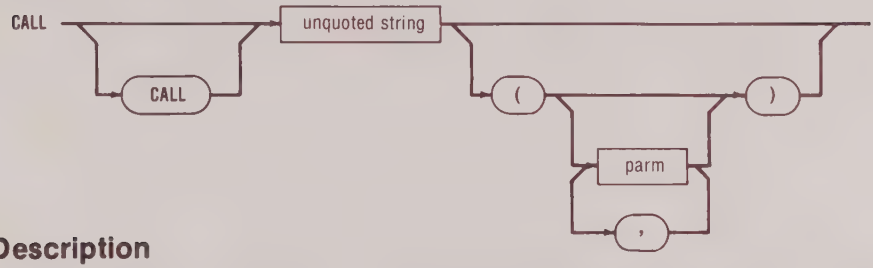
```
100 BREAK 1%          !send break to kb1:
```

Statement

Usage

CALL {unquoted string} (optional parameter list)

Syntax Diagram



Description

The CALL statement executes a subroutine file loaded by the LINK statement.

- The CALL statement can be used in either Immediate or Run mode.
- As shown in the syntax diagram, the CALL statement verb is not required. This is called "an implied CALL". If CALL is not used, then the leading characters of the unquoted string must not conflict with a BASIC verb.
- The unquoted string contains the subroutine name. The subroutine must be present in memory when the CALL statement is executed. If the subroutine is not present, error 705 (Call to undefined subroutine) is displayed.
- Zero (0) to 10 parameters can be passed to a subroutine as part of a CALL statement.

    1. Parameters can be:

        a. Variables

        b. Constants

        c. Expressions

        d. Arrays (but not virtual arrays)

        e. Individual array elements

2. The parameter list must be enclosed in parentheses.

3. Individual parameters must be separated by commas.

4. The required parameter format is described in the Subroutine section of this manual.

CHR$() Function

11

Format

CHR$(numeric expression)

Description

The CHR$ function retuns as ASCII string character corresponding to the decimal value of an integer or floating-point numeric expression.

- The expression may be either integer or floating point.
- The allowable range of values is –32768 to 32767.
- If the expression has a floating point result, it is converted to an integer by truncation, i.e., the integer value will be the largest integer smaller than or equal to the floating point value.
- When sent to the display via PRINT, the character displayed or the control response corresponds to the integer represented by the lower seven bits of the value of the numeric expression. For example, PRINT CHR$(72) displays the letter H, just as PRINT "H" does.
- CHR$(x) will generate 7-bit ASCII codes corresponding to the decimal column in Appendix G if the numbers 0 through 127 are used.
- To set the eighth bit to 1, add 128 to the value in the decimal column corresponding to the desired character or code pattern.

CHR$()
Function

Example

The following example illustrates the results of common uses of the CHR$ function.

```
1000    A = 64 \ B% = 50% \ C% = 70%
1010    PRINT CHR$ (A)                  ! Displays "@"
1020                                    ! (ASCII value of "@" is 64)
1030    B$ = CHR$ (B%)                  ! Assign "2" to B$
1040                                    ! (ASCII value of 2 is 50)
1050    PRINT CHR$ (55) + CHR$ (C%)     ! Displays "7F"
1060                                    ! (ASCII value of 7 is 55)
1070                                    ! (ASCII of F is 70)
1080    PRINT CHR$ (A + B% + 3)         ! Displays "u"
1090                                    ! (ASCII value of u is 117)
1100    PRINT CHR$ (2880)               ! Displays "@"
1110                                    ! (ignores upper 7 bits)
1120    PRINT CHR$ (72.65)              ! Displays "H"
1130                                    ! (Truncates fraction)
```

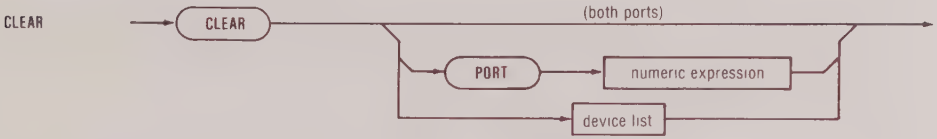
NOTE

Some integers used in the CHR$ function result in display commands when printed and do not appear on the screen. For example PRINT CHR$(7) activates the beeper.

Usage:

CLEAR [device(s)]
CLEAR PORT [number or expression]

Syntax Diagram



Description

CLEAR sends a device clear or selected device clear message to the specified device(s), to all devices on the specified instrument port, or to all devices on both instrument ports.

CLEAR without a device list sends a device clear (DCL) message.

- CLEAR, without a port specified, sends DCL to both IEEE-488 Bus ports.
- To send DCL to only one port, the word PORT must be used, followed by an expression or number that evaluates to 0 or 1.

CLEAR with a device list sends a selected device clear (SDC) message.

- Instruments listed are addressed to be listeners.
- SDC (selected device clear) is then sent to the designated port(s).

Example

The following examples illustrate common uses of the CLEAR statement:

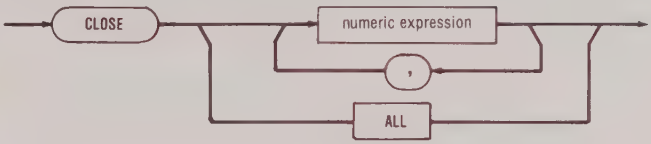
STATEMENT	RESULT
CLEAR	Clear all instruments on both ports.
CLEAR @ 1 @ 102	Clear device 1 on port 0, and device 2 on port 1
CLEAR PORT 1	Clear all devices on port 1

Statement

Usage

CLOSE [numeric expression]
CLOSE ALL

Syntax Diagram



Description

The CLOSE statement frees a previously opened channel for other use. The CLOSE statement requires a comma separated number list as an argument. The CLOSE ALL statement is equivalent to a numeric list of all channels.

As part of this process, some specific actions are taken:

- The input or output of data in memory to or from the specified channel is first completed.
- Interrupts are disabled for the channel if it was opened for input from a serial (RS-232-C) port and interrupts were enabled. This is equivalent to an OFF #n statement in addition to closing the channel.
- An End-of-File mark (CTRL/Z character) is then sent, if the channel was opened for sequential output.
- Makes extra space reserved for a NEW file available again.

CLOSE
Statement

Example

The following example illustrates the effect of reserving more space for a new file than is actually used when the file is closed. If only 25 blocks are actually written to the file when line 510 is executed, the extra five blocks assigned to the file become available as free space on MF0: (i.e., usable by another OPEN command). Only the 25 blocks containing the information written by the program would remain as part of the file RESIST.DAT.

```
20   A$ = "MF0: " \ B$ = "RESIST.DAT"
30   OPEN A$ + B$ AS NEW FILE 5% SIZE 30%
40   ! Other statements
510  CLOSE 5%
```

CMDFILE 14
Function

Format

CMDFILE

Description

An executing BASIC program can determine whether or not it has been called as a result of command file processing by using the CMDFILE function. This function is called as : CMDFILE.

- CMDFILE returns E1% if a command file is active and 0% if no command file is active.

Example

```
1000 ! Subroutine "Delete a file" given in variable "s$"
1010 ! Confirm deletion only if a command file is NOT active.
1020 !
1025 s$ = "foo.bas"
1030 if cmdfile then 1070
1040 print "Really clobber file "; s$;
1050 input line an$
1060 if ucase$(left(an$, 1%)) <> "Y" then 1080
1070 kill s$
1080 return
```

CMDLINE$
System Function

Description

The CMDLINE$ function reads the current FDOS command line and returns a string result. The length of the string is limited to 80 characters, the limit of the FDOS command line.

- Memory overflow is the only error that can result from the use of the CMDLINE$ function.

- The CMDLINE$ function may be used to read arguments from the FDOS command line.

Remarks

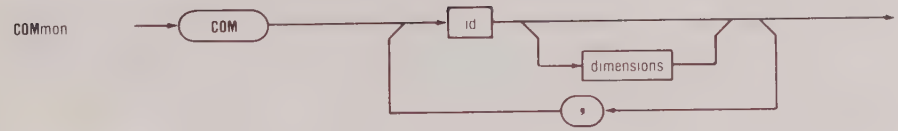
See the entry for the SET CMDLINE$ statement.

Statement

Usage

COM {id list}

Syntax Diagram



Description

COM reserves variables and arrays in a COMmon area for reference by chained programs. COM arguments are valid BASIC array, integer, and floating-point variable names in a comma separated (",") list. Array variables must include their size declaration.

- Only floating-point and integer variables may be stored in the common area.
- String variables may not be stored in the common area.
- String variables may be stored in virtual arrays for access by chained programs. This technique is discussed in Section 7: Data Storage.
- All programs accessing a common area must use COM statements that are identical in order, type, and array sizes; the actual variable names, however, may be different.

Example

For example, assume that a chained program requires the use of three floating point simple variables, an integer simple variable, a floating-point array, and an integer array defined in a previous program. The first program could use a COM statement such as:

```
10      ! Program A
20      COM  A, B, C, F%, D(24%), T%(100%)
        .
        .
        .
1050    RUN "B"
1060    END                              ! End of program A
```

The second program could then use:

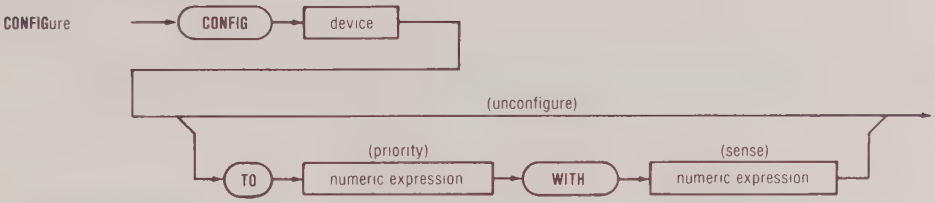
```
10      ! Program B
20      COM L1, L2, L3, G%, K(24%), P%(100%)
        .
        .
        .
```

Note that while the names of the variables stored in the common area have changed between programs, the order and type of the variables are exactly the same.

Usage

CONFIG [device TO priority WITH sense]

Syntax Diagram:



Description

CONFIG (CONFIGure) will either configure or unconfigure an instrument for parallel poll.

- TO specifies the DIO line on which the instrument should respond to a parallel poll.
- The WITH clause specifies the active sense (0 or 1) the instrument should use in responding to the poll.
- If the 'TO line WITH sense' clause is omitted, a PPD (Parallel Poll Disable) message will be sent to the instrument.
- Several instruments may be configured to respond with the same sense (1 or 0) on the same DIO line.
- If the sense is 1, then their poll bits are logically ORed.
- If the sense is 0, then their poll bits are logically ANDed.
- Use the OR configuration to determine whether any instrument is busy or needs service.
- Use the AND configuration to determine if all instruments are busy or need service.

CONFIG
Statement

Example

The following example configures device 4 on port 0 to respond on DIO2 with a 1 as affirmative poll response.

```
14070   CONFIG @ 4 TO 2 WITH 1
```

The following example configures device 21 on port 1 to respond on DIO7 with a zero as affirmative poll response.

```
26480   CONFIG @ 121 TO 7 WITH 0
```

The following example disables device 4 on port 0 from responding to parallel polls.

```
580     CONFIG @ 4
```

Remarks

Compare the CONFIG statement with the PPL Function.

CONT TO
Immediate Mode Command

Usage

CONT
CONT TO {linenumber}

Syntax Diagrams

Description

The Immediate Mode CONT TO line number command causes program execution to continue from a breakpoint stop caused by STOP, STOP ON, CONT TO, or ⟨CTRL⟩/C.

- The CONT TO line number command is available only in Immediate Mode.
- CONT TO line number must first be enabled by a breakpoint stop in a running program, caused by STOP, STOP ON, CONT TO, or ⟨CTRL⟩/C.
- Any subsequent action other than entering the CONT TO line number command disables the command. The program must then be rerun to the breakpoint.
- Program execution continues at the statement following the last statement executed.
- When TO line number is included, program execution again stops if the specified line number is encountered, and the statement is not executed.
- CONT TO can be used instead of or in addition to STOP ON and CONT to move quickly through a subroutine or loop that has already been confirmed during Step Mode logic debugging.

CONT TO
Immediate Mode Command

Example

The following example program is in main memory during the interaction that follows.

```
10      FOR I% = 1% TO 2%
20      PRINT I% + I%
30      NEXT I%
40      PRINT "Done!"
50      END
```

With the above program in main memory, the following sequence of commands and RETURN entries would produce the responses shown. Programmer entry is shown to the left, and Controller response is shown to the right.

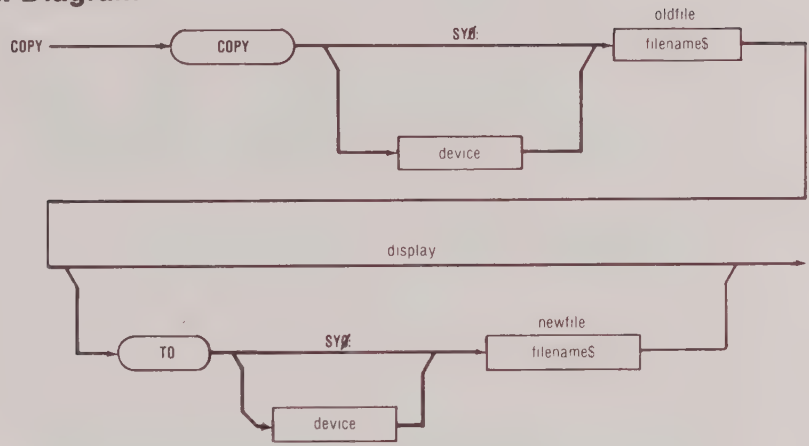
PROGRAMMER ENTRIES	CONTROLLER RESPONSES
STOP ON 10	Ready
RUN	Ready
	Stop at line 10 Ready
STEP	Stop at line 10 Ready
(RETURN)	Stop at line 20 Ready
(RETURN)	2
	Stop at line 30 Ready
(RETURN)	Stop at line 20 Ready
(RETURN)	4
	Stop at line 30 Ready
(RETURN)	Stop at line 40 Ready
(RETURN)	Done!
	Stop at line 50 Ready
STOP ON 10	Ready
RUN	Stop at line 10 Ready
CONT TO 40	2 4
CONT	Stop at line 40 Ready
	Done! Ready

Statement

Usage

COPY {oldfile$} [TO newfile$]

Syntax Diagram



Description

The COPY statement will copy a file either to the screen or to another file.

- For any oldfile$ or newfile$ the device "SY0:" is assumed if no device is specified.
- The extension ".BAS" is supplied if no filename extension is specified.
- If the TO clause is omitted, the input file is copied to the screen.
- If both the input and output files reside on file-structured devices, a block-mode (binary) transfer is performed, which permits binary files to be copied. If this is not the case, the file is assumed to be an ASCII stream (having a line structure and terminated by a ⟨CTRL⟩/Z character), in which case the transfer is performed line-by-line.
- Almost any I/O error may occur when this statement is executed. In addition, if any of the filename expressions are not strings, error 312 (illegal filename) will be reported.

COPY
Statement

Examples

```
copy "sort"
copy "b1.dat" to nf$
copy "kb1:" to "new.dat"
copy "mf1:" + b2$ + ".cil" to "sd0:prog.cil"
```

NOTE

If a COPY from device KB0: (the console keyboard) is performed, the keyboard is immediately placed in the ECHO mode (see the SET ECHO and SET NOECHO statements).

COS() Function

Format

COS (angle in radians)

Description

The COS function returns the cosine of an angle that is expressed in radians.

- COS has a floating-point number as an argument, and returns a floating-point number.
- The range of input values is between and including the limits of ±32767 radians (10430.06 pi radians). Error 607 results from input values outside these bounds.
- The range of output values is between and including the limits of +1 and — 1.
- Input values within approximately 1E-16 of integer multiples of PI/2 give a result of zero, rather than underflow error.

Example

```
Ready
print cos(45)
 0.525322

Ready
```

Format

CPOS(R,C)

Description

The CPOS function returns a string which, when sent to the display with the PRINT statement, positions the display cursor to the row and column coordinates given as arguments.

- CPOS has two numeric expressions (R and C) as arguments and yields an 8-character string. The string is always eight characters long, in the form: ESCape [row ; column H. For example, CPOS(3,20) is equivalent to CHR$(27)+"[03;20H".
- Both numeric expressions may use integer or floating-point numbers.
- If the expression for line or column is floating point, it must be within the range of integers, and will be truncated to an integer.
- The domain of acceptable input is –32768 to 32767.
- The range of row numbers acceptable to the display is 0 to 16 in normal display mode, and 0 to 8 in double size display mode.
- The range of column numbers acceptable to the display is 0 to 80 in normal display mode, and 0 to 40 in double size display mode.
- If either the line or column is less than zero, a value of zero is assigned. Thus display positions 0 and 1 are identical.
- If the line or column is greater than 99, a value of 99 is assigned.
- Numeric strings in the result are two characters in length. Numbers between 0 and 9 are given a leading 0.
- Additional limits are imposed by the video display module:

  1. A line or column number of zero is interpreted as one.
  2. A line number greater than 16 is interpreted as 16.
  3. A column number greater than 80 is interpreted as 80.
  4. In double-size display mode, these limits are respectively line 1, line 8, and column 40.

CPOS()
Function

Examples

The following examples illustrate the results of different uses of the CPOS function.

STATEMENT	RESULTS
`PRINT CPOS (2, 4);`	Place the cursor at row 2 column 4.
`PRINT CHR$(27); "[02;04H";`	Place the cursor at row 2 column 4. This is identical to the previous example.
`A$ = CPOS(2, 4)`	Place the string "ESCape [02;04H" into A$.
`PRINT CPOS(A, B); "HELLO"`	Displays "HELLO" at row A, column B.
`PRINT CPOS(A+5, B); "HELLO"`	Displays "HELLO" at row A+5, column B.

The CPOS function may be assigned to a string variable or added to strings for display formatting. It may take various forms as shown in the following examples which display "Fluke Instrument Controller" at line 10, column 30.

Example 1:

```
10   A$ = CPOS(10,30)
20   B$ = A$ + "Fluke Instrument Controller"
30   PRINT B$
```

Example 2:

```
10   PRINT CPOS(10,30); "Fluke Instrument Controller"
```

The CPOS string function may also be used for more creative displays. The following example displays a scaled SIN function, using CPOS:

```
10 ! *** Display Sine Function ***
20 !
30 FOR X = 1 to 80                  !Setup loop for display length
40 Y = 6 * SIN (2 * PI * (X/40)) + 9 !Compute scaled sine function
50 PRINT CPOS(Y,X); '*';            !Position cursor and display *
60 NEXT X                           !Loop
```

The following example illustrates a method of causing the cursor to disappear during a double-size message display. The technique is to move the cursor alternately to the lower left and lower right corners. Normally, the ON KEY interrupt would be enabled (discussed later in this section), allowing operator input to break the loop:

```
1000 PRINT CPOS(8,1)
1010 PRINT CPOS(8,40)
1030 GOTO 1000
```

Usage

DEF FN[function name] (argument list)

Syntax Diagram

DEFine

Description

The DEF FN statement allows functions to be defined for subsequent use in expressions.

- The function name can be any variable name, such as A%, FA$, Z, including variables previously used in the program. (e.g., FNA% does not conflict with A% already existing in the program.)
- The argument named must be a simple variable name (i.e., not subscripted).
- The argument name is local to the definition. This means that the same variable name used elsewhere in the program does not affect and is not affected by the definition of the function or by the function's execution.
- An argument name must be included in the function definition even if it is not used.
- The argument name is enclosed in parentheses following the name of the function.
- Multiple arguments are not allowed.

DEF FN
Statement

Example

The following example defines user function A to be three times the argument passed to the function, designated as X. After the function has been defined it may be used anywhere in the program just as if it were a system function, i.e., as a part of an expression.

```
10 DEF FNA(X) = 3 * X          ! Defines function as 3 * X
20 X = 5                       ! Assigns 5 to X
30 PRINT FNA(7)                ! Displays 21 (3 times 7),
40 REM                         ! not 15 (3 times X)
```

The following examples illustrate the use of a system function and the use of a system constant as part of the definition of a user function. In this example, user function C returns the arcsine of the input value and (line 200), user function R converts the input value from radians to degrees. The remainder of the program gets an input value, finds the arcsine. converts the result to radians, then prints the answer.

```
100 DEF FNB(C) = ATN (C / SQR (1 - C * C)) ! define function B
200 DEF FNR(R) = R * (180 / PI)            ! define function C
300 PRINT "value to convert "; \ INPUT X   ! get data
400 A = FNB(X)                             ! calc arcsine
500 B = FNR(A)                             ! convert to degrees
600 PRINT "The arcsine of "X;" is ";B;" degrees." ! answer
700 END
```

Sample program run:

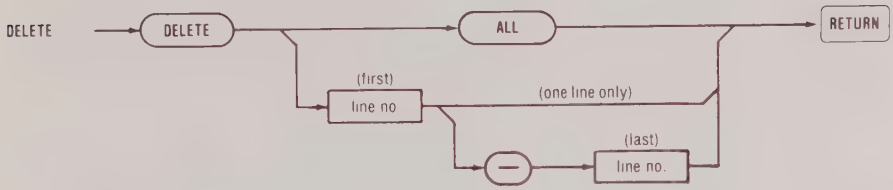
```
Ready
Run
value to convert ? .5
The arcsine of .5 is 30 degrees.

Ready
```

Immediate Mode Command

Usage

DELETE {ALL}
DELETE {linenumber}
DELETE {from linenumber - to linenumber}

Syntax Diagram



Description

The DELETE command deletes part or all of a program from memory.

- The entire program is deleted when ALL is specified.
- DELETE ALL also deletes Common Variables (see the COM statement in the Program Chaining section).
- One line is deleted if a single line number is specified. The command is ignored if the line does not exist.
- One line may also be deleted by typing the line number only, followed by pressing RETURN.
- The portion of the program between and including specified lines is deleted if two line numbers are specified.

DELETE
Immediate Mode Command

Examples

The following examples illustrate common uses of the DELETE command.

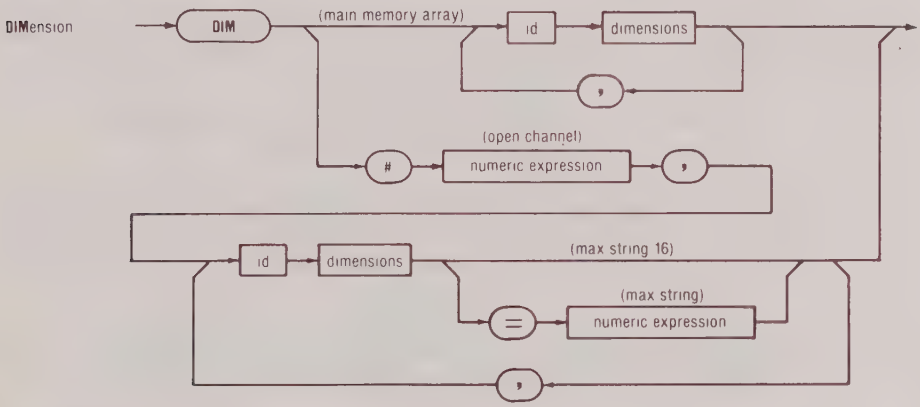
COMMAND	RESULTS
DELETE	No action
DELETE ALL	Deletes entire program and the COM variables
DELETE 100	Deletes only line 100
DELETE 200-300	Deletes lines 200 through 300
400 RETURN	Deletes line 400

Statement

Usage

DIM [id(dimensions)]
DIM #[open channel],[id(dimensions)]
DIM #[open channel],[id(dimensions) = length]

Syntax Diagram



Description

The DIM statement reserves memory or file space for arrays in main memory or on a file-structured device. The DIM statement has the following characteristics when used for a main-memory array.

- An array is a set of variables.
- Each variable is an "element" of the array.
- The maximum array index for each dimension may be specified as one less than the required number of elements, as element counting begins with zero.
- One or two dimensions may be specified.
- A two-dimensional array may be considered as a matrix with the first dimension representing rows and the second dimension representing columns.
- A single DIM statement may dimension one or more arrays, separated by commas.

DIM
Statement

The DIM (DIMension) statement may also be used to assign a previously opened channel to a virtual array and informs BASIC about data organization within the file. Virtual arrays are stored on a file-structured device and are treated as random-access files. The DIM statement has the following characteristics when used to describe a virtual array.

- The # character and numeric expression following DIM specify an open channel from 1 to 16.
- String array declarations may specify the maximum length, in characters, of each element string.
- This length specification follows the array identifier and array size specification. For example, DIM #4, Q$(63%, 63%) = 8% declares Q$ to be a virtual array, through channel 4, containing 64 X 64 string elements of 8 characters each.
- String element lengths in virtual arrays must be a power of 2 between 2 and 512 (2, 4, 8, 16, 32, 64, 128, 256, or 512).
- 16 characters per string is assumed when no length is specified.
- The virtual array DIM statement does not initialize string or numeric variables to nulls or zeros as does the ordinary DIM statement. For this reason, a value must be assigned to virtual array elements before they can be used as source variables. After being dimensioned they contain whatever bit patterns are in their respective disk storage areas.

Examples

Main Memory Arrays

The following examples illustrate some common uses of the DIM statement, with comments on the results of each statement.

	MEANING
DIM A(5)	Dimensions a six-element one-dimensional floating point array with the following elements:

```
A(0)
A(1)
A(2)
A(3)
A(4)
A(5)
```

DIM A%(2, 3) Dimensions a twelve-element two-dimensional integer array with 3 rows (0-2) and 4 columns (0-3):

```
A%(0, 0)    A%(0, 1)    A%(0, 2)    A%(0, 3)
A%(1, 0)    A%(1, 1)    A%(1, 2)    A%(1, 3)
A%(2, 0)    A%(2, 1)    A%(2, 2)    A%(2, 3)
```

DIM OP$(5) Dimensions a six-element string array that could be used to store operator messages.

DIM A(5), A%(2,3), OP$(5)

This statement accomplishes the tasks of the three previous examples in one statement.

Virtual Arrays

In the following example, line 10 specifies that the virtual array file "RESULT.VRT" will be 20 blocks long and accessible on channel 1. Line 20 assigns four virtual arrays to the open channel 1.

```
10   OPEN "RESULT.VRT" AS NEW DIM FILE 1 SIZE 20
20   DIM #1, A$(63%) = 128%, C$(31%), B(40%), A%(500%)
```

Note that the data will fit in 20 blocks, but not in 19:

ARRAY	# OF ELEMENTS	SIZE	# OF CHARACTERS
A$	64	128	8192
C$	32	16	512
B	41	8	328
A%	501	2	1002

TOTAL: 10034 bytes (or characters) dupl

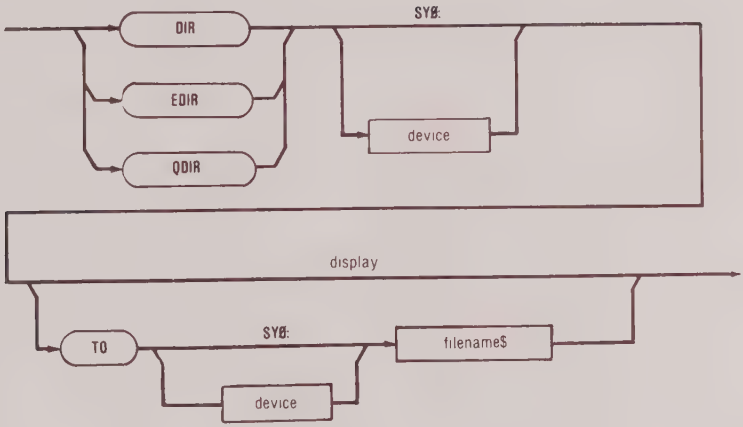
19 blocks = (19 * 512) = 9728 characters
20 blocks = (20 * 512) = 10240 characters

Statement

Usage

DIR [device$] [TO filename$]

Syntax Diagram



Description

The DIR statement prints the directory of a file-structured device.

- Device$ is the name of the device for which a directory listing is desired (which is "SY0:" if omitted), and filename$ is the name of a file to which the directory listing is to be sent (which is "KB0:" if omitted).
- The format of the output resembles a "/L" directory listing in FUP. The format of the output is:

  Directory of XXX: on dd-mm-yy at hh:mm:ss
  Name .Ext Size Prot Date
  xxxxxx.xxx n [*] dd-mm-y

  Total of n blocks in n files, n free blocks.

- Note that I/O errors may be reported by this statement.

Examples

The following examples represent typical uses of the DIR statement.

```
dir
dir "mm0:"
dir dv$ to "kb2:"
```

DISABLE 26 Statement

Usage

DISABLE

Syntax Diagram

Description

The DISABLE statement disables all interrupt types (except ON ERROR and ON CTRL/C) until a corresponding ENABLE statement is executed.

DISABLE CMDFILE Statement

Usage

DISABLE CMDFILE

Syntax Diagram

Description

The DISABLE CMDFILE suspends command file execution to permit input from the keyboard.

- An INPUT statement is used to retrieve input from the user rather than from the command file.
- The Disable CMDFILE can also be used to permanently suspend command file execution because of a program detected error.

Examples

The ENABLE CMDFILE statement "undoes" the effect of DISABLE CMDFILE. For example, to retrieve a line of keyboard input whether or not a command file is active may be done as follows:

```
10 print "What is the name of the dump device";
20 disable cmdfile
30 input line dv$
40 enable cmdfile
```

To stop command file execution, a program may use the following scheme:

```
10 !
20 !  Error exit section -- disable command file and exit.
30 !
40 print "An "; er$; "error has occurred."
50 disable cmdfile
60 exit
```

Remarks

The ENABLE CMDFILE is the logical complement of the DISABLE CMDFILE statement.

DUPL$() Function

Format

DUPL$(string$, count%)
DUPL$(val%, count%)

Description

The DUPL$() function provides a means to create a string of zero or more duplicated characters or strings.

- DUPL$() requires an integer argument (count%) and returns a string of zero or more duplicated characters or strings (string$).
- If count% is a floating point value it will be truncated to integer. This operation may cause an arithmetic overflow (error 601). If the value of count% is less than equal to zero, DUPL$() will return a null string result.
- If the value of string$ is the null string, DUPL$() will return a null string result.
- When string$ is the first parameter, count% gives the number of duplicates of string$ to be produced. Note that this implies that if string$ contains n characters, the length of the result returned by DUPL$() will be n * count% characters long.
- When val% is the first parameter, val% is converted to a one character string in the same manner as the CHR$() function would convert it to a string.
- The value of val% is truncated to use only the low-order eight bits of the integer value.
- If val% is expressed as a floating point quantity, the value will be truncated to integer which may cause an arithmetic overflow (error 601).

DUPL$()
Function

Example

The principal value of DUPL$() is to initialize strings, print banners, create character graphics strings, and to optionally pad strings with a certain character. Printing a 4-character hexadecimal value (with leading zeroes) could be accomplished with DUPL$() as follows:

```
10 print "Value";
20 input i%
30 s$ = rad$(i%, 16%)
40 print "Hex "; DUPL$("0", 4% - len(s$)); s$
```

Which produces:

```
Ready
Run
Value? 15
Hex 000F

Ready
```

Note that line 40 in the program shown above could also have been written as:

```
print "Hex "; dupl$(48%, 4% - len(s$)); s$
```

or as:

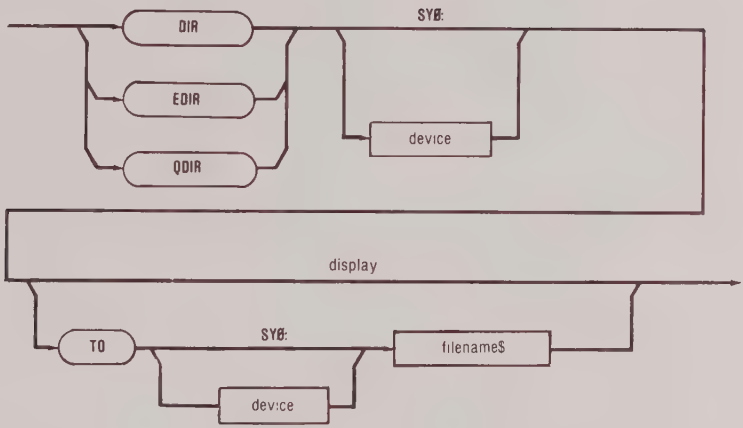
```
print "Hex "; dupl$(ascii("0"), 4% - len(s$)); s$
```

Statement

Usage

EDIR [device$] [TO filename$]

Syntax Diagram



Description

The EDIR statement prints the directory of a file-structured device.

- Device$ is the name of the device for which a directory listing is desired (which is "SY0:" if omitted), and filename$ is the name of a file to which the directory listing is to be sent (which is "KB0:" if omitted).
- The format of the output resembles a "/E" directory listing in FUP.
- An unused (or empty) directory entry is listed as

  ⟨not used⟩ nnn

  (in which "nnn" is the size in blocks).
- A tentative directory entry (a "new" file which has been opened but never closed) is listed as

  ⟨temp ent⟩ nnn

  (in which "nnn" is the size in blocks).

EDIR
Statement

- The format of the output is:

```
Directory of XXX: on dd-Mmm-yy at hh:mm:ss
 name .ext       size    prot date
 xxxxxx.xxx      nnn     [*]   dd-Mmm-yy
 ⟨not used⟩      nnn
 ⟨temp ent⟩      nnn
 Total of n blocks in n files, n free blocks.
```

- Note that I/O errors may be reported by this statement.

Examples

The following examples represent typical uses of the EDIR statement.

```
edir
edir "mm0:"
edir dv$ to "kb2:"
```

Immediate Mode Command

Usage

EDIT [linenumber]

Description

The editor provided as a part of the Fluke BASIC Interpreter program is an easy-to-use character-oriented editor. The Edit Mode allows the user to create, delete, or modify the characters that make up program lines in main memory. Program lines are stored in main memory for subsequent use by other modes. The editing keys in the upper right corner of the keyboard plus the ⟨CTRL⟩/U, BACK SPACE, RETURN, and LINE FEED keys control the cursor and delete text. The remaining keys are used for text entry. This section describes the edit keys and their use along with other editor features.

- Edit Mode is entered from Immediate Mode by typing:

  EDIT ⟨RETURN⟩
  or
  EDIT line number ⟨RETURN⟩

- Editing begins with the lowest numbered line of the program in memory unless another line is specified.

- No line number specification is used when beginning the edit of a new program when no other program is in memory.

- Program entry procedure is the same as in Immediate Mode.

- Immediate Mode commands and program statements cannot be executed while in Edit Mode.

- Exit from Edit Mode to Immediate mode by entering ⟨CTRL⟩/C.

- The special edit keys on the programmer keyboard are enabled.

- Up to 15 lines of the program in memory are displayed, beginning at the first line or at the line number given with the command.

- Edit Mode enables the user to scroll the cursor forward or backward in a program as well as right or left on program lines.

EDIT
Immediate Mode Command

- Edit Mode enables the user to delete characters, portions of lines, or entire lines.

- Edit Mode also enables the user to duplicate entire program lines. The following examples illustrate the two different uses of the EDIT command.

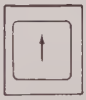
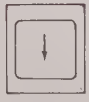
COMMAND	RESULT
EDIT	Select Edit Mode and display up to 15 lines of the program in memory, beginning with the first line. If no program exists, the display is cleared and the cursor is positioned to the upper left corner of the display.
EDIT 1000	Select Edit Mode and display up to 15 lines of the program in memory, beginning with the first line greater than 999. If no program exists or the last line number is less than 1000, the display is cleared and the last line of the program is displayed on the top of the screen.

Edit Mode Keys

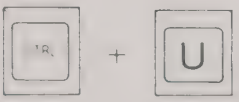
Some of the keys on the programmer keyboard have special functions that are enabled or modified in Edit Mode. Any key, if held down, performs its function repeatedly. The figure below describes the special functions of the Edit Mode Keys.

NOTE

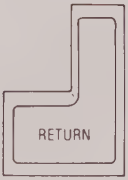
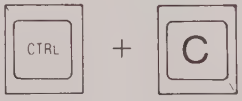
Any edit command that will move the cursor from the current line or ⟨CTRL⟩/C is not accepted if the line does not pass a check for correct syntax. A blinking error message (e.g.: "Mismatched Quotes") will be displayed until the line is corrected.

KEY	ACTION
	Move one position left. Ignored if already at the left margin.
	Move one position right. Ignored if already at the right end of the line.
	Move one position up. If the line above is shorter than the current column position, move left to the end of that line. Scroll down one line if the cursor is on the top line of the display, and another line is available. This action will not be done if the line does not pass a syntax check.
	Move one position down. If the line below is shorter than the current column position, move left to the end of that line. Scroll up one line if the cursor is on the bottom line of the display, and another line is available. This action will not be done if the line does not pass a syntax check.

EDIT
Immediate Mode Command

KEY	ACTION
	Delete from the cursor position to the end of the line. If the cursor is at the left margin, delete the entire line and move the rest of the program up one line to fill the deletion.
	Delete the character at the cursor position and move the remaining characters left one position to fill the deletion. When the key is held down for repeat, the portion of the line to the right of the cursor will progressively disappear.
	Delete the character to the left of the cursor position and move the remaining characters left one position to fill the deletion. When the key is held down for repeat, the portion of the line to the left of the cursor will progressively disappear as the portion to the right moves to the left margin. This key function is also available in Immediate Mode.
	Delete the current line.
	Move to the left margin.

EDIT
Immediate Mode Command

KEY	ACTION
	Move to the right end of the line.
	When the cursor is at the right end of the line, open a new empty line below, and move to its left margin. When the cursor is not at the right end of the line, break the line into two lines. The cursor position identifies the first character of the new (second) line. This action will not be done if the portion of the line that was to the left of the cursor does not pass a syntax check.
Character Keys	Insert characters at the current cursor position. Each character entry moves the cursor right one position along with any text to the right of the cursor. Entries that would result in a line length greater than 79 characters are not accepted, and produce a beep sound.
	Return to Immediate Mode.

EDIT
Immediate Mode Command

Additional Editor Features

It is not necessary to insert a line in correct sequence. Regardless of the order in which program lines are entered, the editor will store them in memory in the correct line number sequence. When the cursor is instructed to move from a line, the editor checks for some syntax errors, such as omitting a quote, parenthesis, or line number. If a line does not pass the check, an error message is displayed and the cursor is not allowed to leave the line until the error is corrected.

If a program line is renumbered by deleting all or part of its line number and then entering a new line number, a duplicate line will result. One line will have the original line number, the other line will have the new line number. This may be seen by scrolling the modified line on and off the display, in EDIT mode.

If ⟨CTRL⟩ / C is entered when the current line will not pass the syntax check, the blinking error message is displayed in Immediate Mode and the line is not stored in memory.

There are many errors the editor will not detect, such as forgetting to dimension an array or specifying GOTO with a nonexistent line number. Such errors will be detected only when the program is run.

The cursor will not scroll above the lowest line number nor below the highest line number in the program. If the cursor is in the middle of the program and a new last line is entered at that position, the cursor will not scroll down past that line. To correct this condition:

- The cursor may be scrolled in the opposite direction until the line entered out of sequence disappears from the display. Reverse scroll direction again and the line will then be in proper sequence.

- Type ⟨CTRL⟩ C, and then type EDIT, followed by the line number that needed editing. Lines will then be displayed in correct sequence, allowing access to all lines.

EDIT
Immediate Mode Command

The editor stores program lines in memory with the line number shown on the display. If any other program line has the same line number as that shown on the display, it is replaced with the contents of the line shown on the display. This feature can be used to duplicate program lines by changing only the line number and moving the cursor off the line. The line with the previous line number is not deleted by this process. The display, however, will show only the most recent line number entered. To see both resulting lines, scroll the entered line off the display and back on.

When a line is scrolled off the display with the same line number as a line previously stored, the original line in memory will be replaced by the one which is scrolled off. In order to prevent this from occurring, assign each line a unique line number.

ENABLE 3.1 Statement

Usage

ENABLE

Syntax Diagram

Description

The ENABLE statement re-enables all interrupts previously turned off by means of the DISABLE statement.

ENABLE CMDFILE 32
Statement

Usage

ENABLE CMDFILE

Syntax Diagram

Description

The ENABLE CMDFILE statement re-enables command file execution after a previous DISABLE CMDFILE statement.

Examples

The ENABLE CMDFILE statement "undoes" the effect of DISABLE CMDFILE. For example, to retrieve a line of keyboard input whether or not a command file is active may be done as follows:

```
10 print "What is the name of the dump device";
20 disable cmdfile
30 input line dv$
40 enable cmdfile
```

To stop command file execution, a program may use the following scheme:

```
10 !
20 !  Error exit section -- disable command file and exit.
30 !
40 print "An "; er$; "error has occurred."
50 disable cmdfile
60 exit
```

Remarks

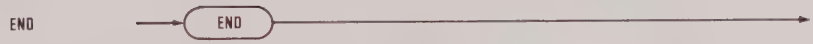
The DISABLE CMDFILE statement is the logical complement of the ENABLE CMDFILE statement.

Statement

Usage

END

Syntax Diagram



Description

The END statement is used to indicate the absolute end or the end of the main body of a program.

- Subroutine code may follow the END statement.
- Code following END will not be executed unless it has been accessed by a GOSUB or GOTO (or implied GOTO via IF...THEN with a line number).
- END is optional.
- Executing the END statement causes an immediate return to the Immediate mode.

Example

The suggested use of this statement is to designate the physical end of your program:

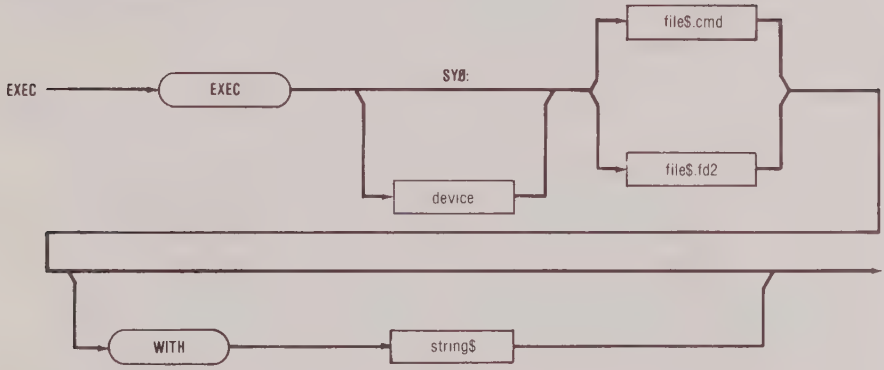
```
32767 END
```

Statement

Usage

EXEC filename$ [WITH command$]

Syntax Diagram



Description

The EXEC statement permits a BASIC program to chain to a machine language program or to a command (".CMD") file.

- The EXEC statement is similar to the RUN statement (in that the program that executes the statement is terminated immediately).
- File$ is the name of the the executable or command file to be executed.
- The optional WITH clause specifies a new command line to be passed to the new program.
- The string command$ will act as though it had been entered following the file name in a command to FDOS.
- If no device is specified, the device "SY0:" will be assumed.
- The only extensions permitted with this statement are ".CMD" and ".FD2". If no extension is specified, a file with the extension ".CMD" will be searched for first; if no ".CMD" file is found, the BASIC system will then look for a ".FD2"File with that name.

EXEC
Statement

- The command line argument plus the length of the file name (less any ".CMD" or ".FD2" extension) may not exceed 80 characters. This limitation is imposed by FDOS. The actual command line passed to the EXEC'ed program will be file$ (without any device name or extension), and, if command$ is specified, a space and the string given as command$. The entire line is terminated by a carriage return character.

Example

BASIC STATEMENT	FDOS EQUIVALENT
`exec "edit" with "test.bas"`	`sy0:edit test.bas`
`exec "mf1:fup"`	`mf1:fup`
`exec "mf0:fup"`	`mf0:fup`
`exec "edit.fd2" with "foo"`	`sy0:edit foo`

Statement

Usage

EXIT

Syntax Diagram

Description

The EXIT statement causes an immediate termination of program execution. The controller then loads and executes the program specified in the SET SHELL statement or FDOS.

The EXIT statement may also be given in the Immediate Mode. The EXIT statement, when given in the Immediate Mode, will ask the user if he really wants to exit, before actually exiting, if the program currently in main memory has been edited but not SAVED or RESAVE'd (SAVEL or RESAVEL, also).

Example

```
Ready
old "test"

Ready
5 print "hello"
exit
Program has been changed but not saved. Really EXIT? n
```

Format

EXP (numeric expression)

Description

The EXP function returns a floating-point number equal to the result of raising the number e to the power equal to the given input value.

- EXP has a floating-point number as an argument, and returns a floating-point number.
- The value used by the Instrument Controller for e is 2.71828182845905
- The domain of input values is negative numbers, zero, and positive numbers up to and including 708. Error 605 results when an input of 709 or greater is used.
- Input values of –709 or less produce a result of 0.

Example

```
1010 PRINT EXP(21)        ! Display e^21
```

FLEN 37
System Variable

Description

FLEN is a system variable set to the length, in blocks, of the last file opened on a file-structured device.

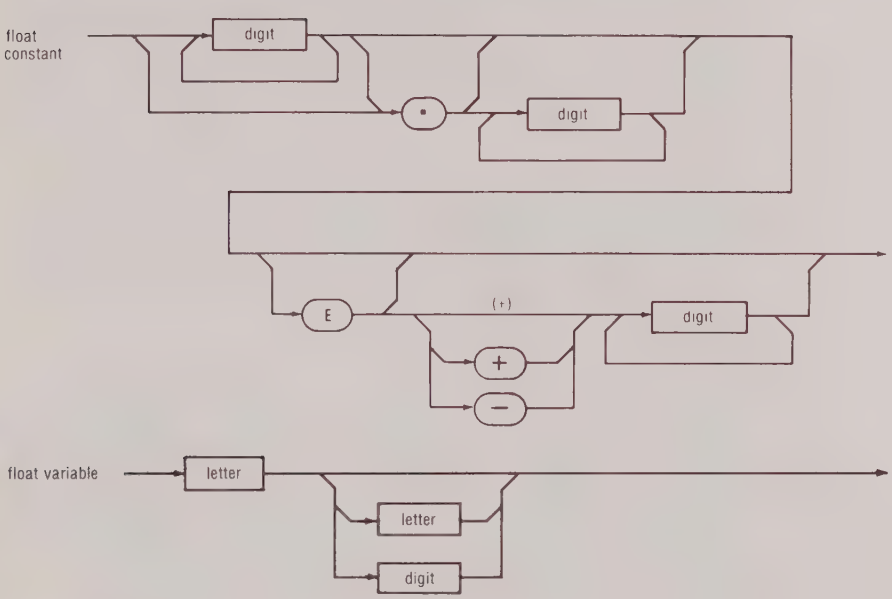
- FLEN may be used with OPEN to check the number of blocks in an open file before attempting to use a new file.
- One or more open files may need to be closed and perhaps deleted (see KILL) before opening a new file if the disk memory available is not sufficient to hold all opened files in contiguous areas.

Example

In the following example, channel 1 is closed only if the last file opened has a length greater than five blocks. This may be done to limit open files and conserve disk space.

```
IF FLEN)5% THEN CLOSE 1%
```

Syntax Diagram



Description

Floating-point data has the following characteristics:

- Decimal exponent range: +308 to –308.
- Exact range: 2.2225074E-308 to 3.595386E+308.
- Resolution: 15 decimal digits.
- Inexactness in the numeric representation.
- Memory requirement (per data item): 8 bytes.
- Represented internally in binary in accordance with proposed standard "IEEE Floating-Point Arithmetic for Microprocessors". Copies of this standard are available from The Institute of Electrical and Electronic Engineers, 345 East 47th Street, New York, New York, 10017.

FLOATING-POINT
Variables

Unless modified by a PRINT USING statement, floating-point data is displayed with a leading space or sign and a trailing space. It is printed out to seven significant digits. A value from .1 to 9999999 inclusive is printed out directly. A number less than .1 is printed out without E notation if all of its significant digits can be printed. All other values are printed in E notation (+0.dxxxxxxE+yyy), where d is a non-zero digit, x is any digit, and trailing zeros are dropped.

NOTE

Floating-point numbers may be displayed with up to 15 significant digits by using the PRINT USING Statement. Use a format specifier that represents the number of significant digits that you want displayed.

Floating-Point Constants

Floating-point constants, often called real numbers, are represented in a program in decimal or possibly scientific notation. The syntax diagram illustrates the proper representation of floating-point numbers. A number in scientific notation, with an exponent following "E", represents a number multiplied by a power of 10. Examples of floating-point constants are:

.005	
6354.33	
-134.7	
-12E2	Represents -1200
0.13E-05	Represents .0000013
0.1E6	Represents 100000
.1E-400	Floating-point number outside the legal range. Returns error 602.

Floating-Point Variables

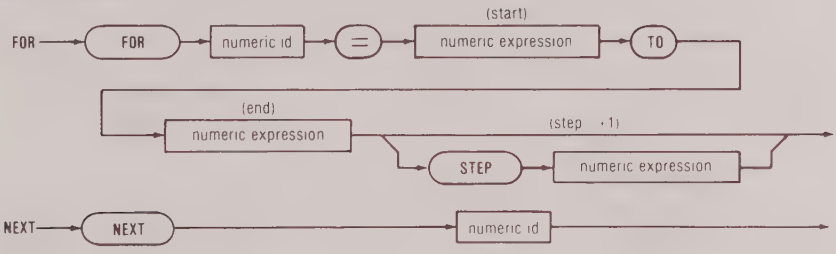
Floating-point variables are designated by a letter followed by an optional second character. The second character can be a letter or a number. The following variable names are not allowed, since they are keywords of Fluke BASIC: AS, FN, IF, LN, ON, OR, PI, TO.

FOR and NEXT Statements

Usage

FOR {index} = {begin} TO {end} [STEP {step}]
    program steps
    more program steps
    etc...
    NEXT {index}

Syntax Diagram



Description

The FOR statement sets up a loop which repeatedly executes the statements contained between the FOR statement and the NEXT statement.

- There must not be a FOR without a NEXT, or a NEXT without a FOR.
- The FOR statement specifies the number of times the loop is to be repeated by specifying limit points and step increment of the index.
- If no step increment is specified, step +1 is assumed.
- The index must be numeric.
- The index must not be an array element.
- When a FOR statement is initially encountered, the index value is compared to the limit. If the index is past the limit, the FOR-NEXT loop is not executed.
- The NEXT statement compares the index with the limit after incrementing it.

FOR and NEXT
Statements

- When two or more FOR-NEXT loops are nested, each FOR statement must have a corresponding NEXT statement. Furthermore, the innermost FOR statement must match the innermost NEXT statement, and so on, to the outermost FOR-NEXT statement pair.

Examples

The following examples compare two different ways of constructing program loops, using the FOR - NEXT, and using IF - GOTO. Note the different comparisons at line 100, depending on the sign of the STEP.

FOR - NEXT

```
10 FOR I = 1 TO -5 STEP -2
20! Other statements
30!
100 NEXT I
110 ! Other statements

10 FOR J = 10 TO 100 STEP 20
20! Other statements
30!
100 NEXT J
110 ! Other statements
```

IF - GOTO

```
10 I = 1
15 IF I < (-5) GOTO 110
20! Other statements
30!
100 I=I + (-2) \ GOTO 20
110 ! Other statements

10 J = 10
15 IF J > 100 GOTO 110
20! Other statements
30!
100 J=J + 20 \ GOTO 20
110 ! Other statements
```

The following example loops five times, and then prints the index value. Note that the index has incremented to six.

```
10  FOR I%=1% TO 5%                  ! Increments by 1 through loop
20  PRINT I%                         ! Display value of I%
30  NEXT I%                          ! Repeat loop until F=5%
40  PRINT "Index Value is";I%
```

The display will show:

```
Ready
run

 1
 2
 3
 4
 5
Index value is 6

Ready
```

The following example illustrates what happens when the index never equals the final value.

```
10  FOR A% = C% TO D% STEP 2%       ! C% = 5 and D% = 10
20  PRINT A%                        ! Display value of A%
30  NEXT A%                         ! Increments A% by
40  REM                             ! Loops until A% = 10
```

The display will show:

```
Ready
run
5
7
9

Ready
```

If line 10 used C and D rather than C% and D%, and if C = .6 and D = 10.6, the display would show:

```
Ready
run

1
3
5
7
9

Ready
```

because real values are rounded before assignment to an integer variable.

FOR and NEXT
Statements

The following example illustrates the nesting of two FOR - NEXT loops. Note that one blank line occurs between first and second display lines. This happens because the last value displayed in the line above did not have 16 columns available, causing a carriage return-line feed in the display (See PRINT command).

```
10 FOR I% = 1% TO 4%
20       FOR J% = 1% TO 5% STEP I%
30       PRINT I%, J%,                    ! Print values and then tab
40         REM                            ! to next column
50       NEXT J%
60       PRINT                            ! Move to the next line
70 NEXT I%
80 REM INDENTING SHOWN FOR CLARITY
```

The display will show:

```
 1   1          1   2          1   3          1   4          1   5

 2   1          2   3          2   5
 3   1          3   4
 4   1          4   5
```

The following example decrements the index by -.1.

```
10  FOR I = 1 TO 0.5 STEP -.1
20  PRINT I,
30  NEXT I
```

The display will show:

```
Ready
run
 1   0.9   0.8   0.7   0.6   0.5
Ready
```

The following example shows what happens when a GOTO statement branches out of a FOR-NEXT loop and returns to the loop beginning.

```
10    for x = 1 to 100000
20    y = y + 1
30    print y
40    goto 10
50    next x
60    end
```

Which results in the following display:

```
1
2
3
4
....   left out for brevity
1300
1301
1302

!Ovf error 0 at line 20

Ready
```

Usage

GOSUB {linenumber}

Syntax Diagram

Description

The GOSUB statement transfers control to the beginning of a subroutine. The RETURN statement terminates the subroutine and returns control back to the statement following the GOSUB.

- BASIC allows nested subroutines.
- Within a subroutine, a GOSUB may call another or the same subroutine.
- A GOTO within a subroutine should not pass control permanently to a program segment outside of the context of the subroutine.
- Any valid BASIC statements may be used in the body of a subroutine. Available memory is the only limit to the length of subroutines.

Example

In the following example the subroutine displays the operator's options and obtains his or her choices. Control is transferred to the subroutine at line 180. The display subroutine calls another subroutine (line 1300) to obtain identification of the selected choice (line 1080). When the RETURN statement at line 1390 is executed, control is transferred to line 1090. Three subroutine options are illustrated, each with different interpretations for GOSUB and RETURN.

The following points should be noted about this example:

- GOTO 31000 causes termination of the test without returning from the subroutine. This normally does not represent good program structure.

GOSUB
Statment

- ELSE 1500 causes control to be transferred to lines 1500 - 1590 which can be considered an extended part of subroutine 1000. The RETURN at line 1590 will cause line 190 to be executed next.

- RETURN at line 1090 will cause immediate termination of the subroutine 1000. Line 190 is executed next.

```
100 ! Other statements
180 GOSUB 1000                              ! Get operator choice
190 ! Other statements
999 STOP
1000 REM -- SUBROUTINE -- Display and Execute Operator Options
1010 PRINT "Enter choice by pressing display"
1020 PRINT\PRINT                            ! Skip 2 lines
1030 PRINT "      [Continue test]"          ! KR% = 1 or 2
1040 PRINT\PRINT\PRINT                      ! Skip 3 lines
1050 PRINT "      [Adjust Instrument]"      ! KR% = 3 or 4
1060 PRINT\PRINT\PRINT                      ! Skip 3 lines
1070 PRINT "      [Terminate Test]"         ! KR% = 5 or 6
1080 GOSUB 1300                             ! Get choice, KR%
1090 IF KR%>2% THEN IF KR%>4% GOTO 31000 ELSE 1500 ELSE RETURN
1300 REM -- SUBROUTINE -- Get response from Keyboard
1310 ! On exit KR% will be the row touched (from 1 to 6)
1320 ! KC% will be the column touched (from 1 to 10)
1330 ! Other statements
1390 RETURN
1500 REM -- SUBROUTINE -- Adjust Instruments
1510 ! Other statements
1590 RETURN
31000 REM -- Termination Routine
31010 ! Place instruments in standby state and save test results
31020 ! Other statements
32767 END
```

Statement

Usage

GOTO {linenumber}

Syntax Diagram

Description

The GOTO statement causes a program to unconditionally branch to the specified line number.

- GOTO must not be used to branch out of a FOR-NEXT loop. A user storage overflow (error 0) will eventually result.

- A GOTO statement may be used to begin running a program, starting with any line number in the program. When a GOTO statement is used to run a program, many of the normal actions of a
- RUN statement are not performed, such as reseting variables and random numbers. This may be useful in program debugging.

Example

The following program example illustrates one use of the GOTO statement. The GOTO statement at line 80 causes line 30 to be executed next:

```
10 REM -- Main Program Routine
20 REM
30 GOSUB 4000                    ! Set-up Sequences
40 GOSUB 5000                    ! Test #1
50 GOSUB 6000                    ! Test #2
60 GOSUB 7000                    ! Test #3
70 GOSUB 8000                    ! Test #4
80 GOTO 30                       ! Start Again
```

Remarks

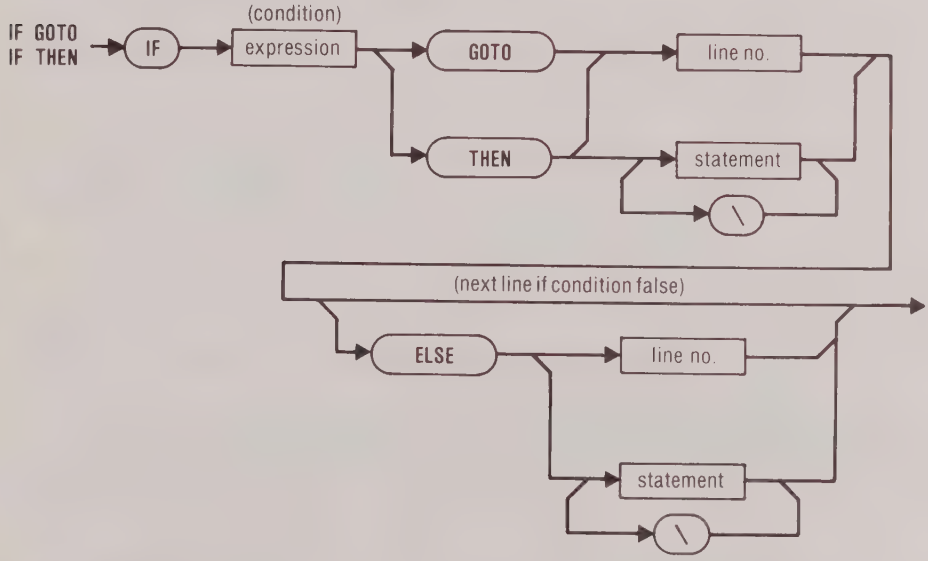
Compare the GOTO statement with the GOSUB statement.

IF-GOTO
IF-THEN
IF-THEN-ELSE
Statement

Usage

IF {condition} GOTO {linenumber}
IF {condition} THEN {BASIC statement}
IF {condition} THEN {BASIC statement} ELSE {BASIC statement}

Syntax Diagram



Description

The IF statement evaluates a condition represented by an expression. The resulting course of action depends on the result of that evaluation and the structure of the statement.

NOTE
Relational expressions such as A=B, A%<3%, A$>=B$ evaluate as -1 if true and 0 if false.

- The conditional expression must result in a value representable as an integer.

IF-THEN-ELSE
Statement

- A non-zero result is TRUE. A zero result is FALSE.
- Control passes to the line number following GOTO, or to the statement or line number following THEN, if the result is TRUE.
- Control passes to the next program line if the result is FALSE and ELSE is not specified.
- Control passes to the statement or line number following ELSE if the result is FALSE and ELSE is specified.
- A line number must follow GOTO, if it is used.
- A line number or a valid BASIC statement must follow THEN, if it is used.
- Multiple statements, separated by the "\" character, may be used after THEN, and after ELSE. Each statement will be done in sequence only if control is passed to that portion of the IF statement as defined above.

Examples

The following examples illustrate the results of various uses of the IF statement:

STATEMENT	RESULTS
IF A THEN 100	If A is non-zero, go to line 100. If A is zero go to the next line.
IF A>B THEN A=B	If A is greater than B, set A equal to B. Then go to the next line.
IF NOT A% GOTO 500	If A% equals -1 (logical true), ignore this statement and go to the next line. Otherwise, go to line 500.

IF-THEN-ELSE Statement

IF A%+B% GOTO 500	If A%+B% is non-zero (A% not equal to –B%), go to line 500. Otherwise go to the next line.
IF A%+B% THEN A=B*5B=10	If A%+B% is non-zero (A% not equal to –B%), assign B*5 to A, and assign 10 to B. Then go to the next line.
IF A% OR B% GOTO 500	If either A% or B% is non-zero, go to line 500. Otherwise go to the next line.
IF A+B=C THEN PRINT C	If A+B equals C, display value of C. Then go to the next line.
IF (1<A) AND (A<6) THEN 450	If the value of A lies within the open interval (1,6) transfer control to line 450.

The following program example shows some common uses of the IF statement. The relational expression in line 140 uses the AND operator to ensure that both conditions are true before transferring control to line 5000.

```
100 REM -- Determine Test to Run
110 PRINT "ENTER UNIT SERIAL NUMBER";    !Print prompt
120 INPUT SN%
130 IF SN% ) 12563% THEN 110             !Equivalent to IF...GOTO 110
140 IF (11000% (= SN%) AND (SN% ( 11500%) GOTO 5000
150 REM -- Run Standard Test
160 ! Other statements
5000 REM -- Run Test for 'Special'
5010 ! Other statements
```

IF-THEN-ELSE
Statement

The following example shows how the IF statement acts on various values. Examining the results will aid in understanding the IF statement. Note that line 100 transfers control back to line 30 except when A% is 0.

```
10 REM -- Illustration of IF with integers
20 PRINT "INTEGER VALUE","ONE'S COMPL.","ODD","DIVISIBLE BY FOUR"
30 INPUT A%
40 PRINT A%, NOT A%,                          !Print values and tab
50 IF A% AND 1% THEN PRINT 'YES;              !Is A% odd?
60 PRINT,                                     !Go to next tab on display
70 IF A% AND 3% THEN PRINT; \ GOTO 90         !Is A% divisible
80 PRINT 'YES';                               !by four?
90 PRINT                                      !Line Feed
100 IF A% THEN 30
110 END
```

Results:

```
INTEGER VALUE    ONES COMPL.     ODD          DIVISIBLE BY FOUR
 1                -2             YES
 4                -5                          YES
 1568             -1569                       YES
 9                -10            YES
-568               567                        YES
-1                 0              YES
-4                 3                           YES
 0                -1                          YES
```

The following example includes the ELSE option. When A is less than 5, B will be set to FNS(A) and the statements following line 200 will be executed until the STOP statement on line 290 is encountered. When A is greater than or equal to 5, B will be set to FNT(A) and the statements following Line 300 will be executed.

```
10 ! Other statements
100 IF A < 5 THEN 200 ELSE 300
200 B = FNS(A)                          ! Use 'S' function
210 ! Other statements
290 STOP
300 B = FNT(A)                          ! Use 'T' function
310 ! Other statements
```

The following example shows the ELSE option followed directly by statements. When A is less than B, J is set to B raised to the power A, and T is set to 0. When A is greater than or equal to B, J is set to the value of A raised to the power B, and T is set to 1.

```
IF A < B THEN J = B^A\T = Ø ELSE
              J = AB\T = 1
```

IF-THEN-ELSE Statement

The following example illustrates nested IF-THEN-ELSE statements. The ELSE is always associated with the closest IF and THEN statement to the left that is not already associated with another ELSE, as indicated by the diagram below the statement. (Note: This statement computes M = minimum of A, B, or C.

```
IF A<B THEN IF A<C THEN M=A ELSE M=C ELSE IF B<C
THEN M=B ELSE M=C
```

The logic is as follows:

If A is less than B, and then if A is less than C, A is the minimum (M).

If A is less than B, and then if A is greater than or equal to C, C is the minimum (M).

If A is greater than or equal to B, and then if B is less than C, B is the minimum (M).

If A is greater than or equal to B, and then if B is greater than or equal to C, C is the minimum (M).

INCHAR() Function

Format

INCHAR(channel%)

Description

The INCHAR() function reads a single character from any open channel or from the keyboard. Channel% is an integer which specifies the channel from which the next character is desired. If the channel number is zero, the next character from the keyboard will be returned.

Character values are returned by INCHAR() as integers, and will have have a value between 0 and 255 (0 and 127 when reading characters from the keyboard).

Use the INCHAR() function when input must be processed on a per-character basis or when data from the keyboard is being entered using NOECHO mode (see the SET NOECHO statement).

- A BASIC program using INCHAR(0%) to read data from the keyboard must be prepared to process line editing (⟨CTRL⟩/U and DELETE) characters as required by the application, since the Operating System may not do so during single-character input.

- No end of file character processing is performed by the INCHAR() function. When reading an ASCII file, the BASIC program must be prepared to process end of file (CHR$(26%)) characters properly.

Example

The example shows a short program segment which gets a single integer value (C%) from channel 1. Line 200 checks to see if the character is the ASCII End-of-File character 26 and branches off to a simple message at line 500 if EOF is found.

```
100 C%=INCHAR(1%)              ! Fetch a single character from CH. 1
200 IF C%=26% THEN 500         ! Pick off End-of-File character
210 PRINT C%                   ! Show the value
.
.
.
500 PRINT "End of file"        ! Print EOF message
510 STOP
```

INCOUNT() Function

Format

INCOUNT(channel%)

Description

The INCOUNT function determines the number of input characters and/or lines available from a serial device (keyboard or RS-232 port).

Channel% is the channel number for which the number of characters available is desired. Channel 0% is the keyboard.

The integer values returned by INCOUNT() are:

0	No characters available.
>0	Number of characters and/or lines available.

Zero will always be returned for an IEEE-488 channel.

An I/O error 308 (channel not open) will be reported if the channel specified is not open (the keyboard -- channel 0% -- is always open). An I/O error 322 (illegal operation for device type) will be reported if the device attached to the channel is not a serial (RS-232 or keyboard) device.

The result returned for the keyboard channel will depend upon the current mode.

- If the keyboard is in NOECHO mode (see the SET NOECHO statement), results returned by INCOUNT(0%) will be reported in terms of characters available.

- If the keyboard is in ECHO mode (see the SET ECHO statement), the result returned by INCOUNT(0%) will be reported in terms of lines available.

- A line is a string of text up to and including an end of line character, CHR$(13).

INCOUNT()
Function

The result for RS-232 channels will be returned as a combined line and character count:

nr of chars	nr of lines

high-byte low-byte

An INPUT statement will be blocked until the "number of lines" value returned by INCOUNT() is nonzero.

An INCHAR function call waits until the "number of characters" value returned by INCOUNT is nonzero.

- Data from RS-232 channels as read by INPUT will be blocked until the number of lines available is nonzero.
- An INCHAR() function is blocked only until the number of characters available is nonzero (the number of lines does not matter).

Example

This example shows how the INCOUNT function can be used to poll an RS-232 port for input before using an INPUT statement to retreive the data. If an INPUT statement had been used without the INCOUNT function, then the program would have been hung up at that point waiting for data.

```
.
.
.
100 C%=INCOUNT(1%)                  ! Poll port 1 for available input
200 IF C% < > 0% AND 255% THEN GOSUB 1000
201                                 ! Branch to input routine if data's
202                                 ! available -- otherwise continue.
.
.
.
1000 ' Input data from the port when it's ready.
1010 INPUT #1, D%
1020 RETURN
.
.
.
```

The following example is a program segment which shows how to isolate the upper and lower bytes from the integer result.

```
200 ! One way to isolate high and low bytes from one integer
210 ! if you need them
220 OPEN "KB0:" AS FILE 1%          ! Take input from keyboard
230 SET NOECHO                      ! Disable echo to screen
240 IN% = INCOUNT(1%)               ! IN% = input char count
250 L% = IN% AND 255%               ! L% = # lines available
260 C% = LSH(IN%, -8%)              ! C% = # chars available ...
```

Line 250 uses a bit mask to isolate the low-order byte of the value returned by INCOUNT(), which is the available line count. Line 260 uses a logical right shift to isolate the high-order byte returned by INCOUNT(), which is the number of characters available.

Statement

Usage

INIT

Syntax Diagram

Description

The INIT (INITialize) statement initializes the specified port (port 0), or all available IEEE-488 ports if not specified. This statement may be used only by the system controller (SYC function).

- INIT places the bus in an idle state.
- INIT sends REN (Remote Enable), IFC (Interface Clear), UNL (Unlisten), UNT (Untalk), and PPU (Parallel POLL Unconfigure) messages.
- REN enables remote programming of instruments.
- IFC unaddresses all listeners and talkers (places all talker and listener functions in an idle state) and terminates all handshakes (places source and acceptor handshakes in either an idle or wait state). IFC also causes any other Controller-In-Charge (CIC) to give up control of the IEEE-488 bus.
- UNL unlistens any listeners (removes instruments from the LISTEN state).
- UNT untalks any talkers (removes instruments from TALK state).
- PPU places devices with remote parallel poll programming capability into idle state. Such devices may be reconfigured for parallel poll with the CONFIG statement described in this section.
- INIT, without a port specified, initializes all available ports.

INIT
Statement

- To initialize only one port, the word PORT must be used, followed by an expression or number that evaluates to the desired port number (0 or 1).

Example

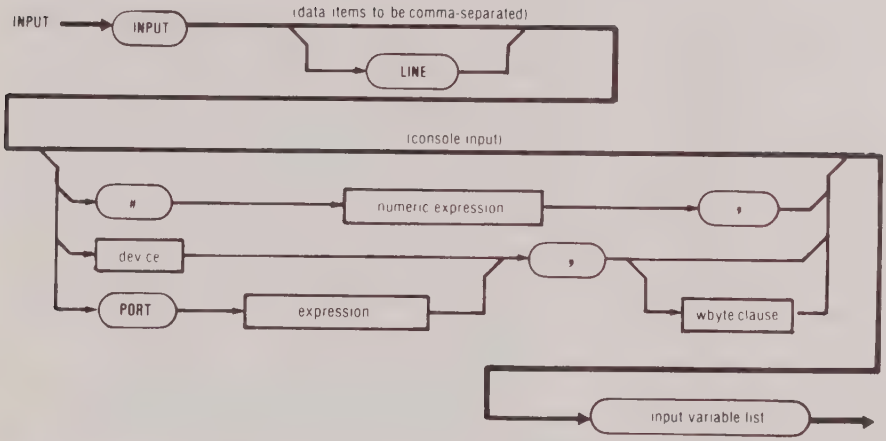
The following examples illustrate uses of the INIT statement:

STATEMENT	MEANING
INIT	Initialize all IEEE-488 interfaces.
INIT PORT 1	Initialize all instruments on the optional IEEE-488 port 1.
INIT PORT A%	Initialize all instruments on the IEEE-488 port identified by the value of A% (must be 0 or 1).

Usage

INPUT

Syntax Diagram



Description

The INPUT statement assigns inputs from devices (either the keyboard or RS-232 devices) or files to variables. This discussion describes keyboard inputs only. A separate description of the INPUT statement as it is used for assigning inputs from external devices follows.

- One or more variables, separated by commas, must follow the INPUT statement.
- INPUT displays a ? character, followed by a space, and halts the program until data is entered and RETURN is pressed.
- If RETURN is pressed without entering anything else, numeric variables are assigned 0 and strings are assigned a null string.
- INPUT assigns keyboard entries, separated by commas, to the variables in sequential order.
- The entry type must match the variable type. String or floating-point data cannot be assigned to an integer variable, for example.

- If an error occurs during assignment, a warning message will be displayed on the screen and a new prompt character, "?", will be given. All variables must then be entered again.
- Strings may be entered in quoted or unquoted form.
- An unquoted string may not contain a comma or begin with a quote.
- Leading spaces in an unquoted string will be ignored.
- A string must be enclosed with quotes to contain commas or leading spaces.
- Internal quotes must be different from enclosing quotes.

Example

The following examples illustrate common uses of INPUT.

INPUT A$ Displays ?. Assigns the entry to A$.

INPUT B%, A$ Displays ?. Assigns the entries (e.g., 1776, GEORGE) as follows: B% = 1776 and A$ = "GEORGE".

In the following example the program will halt until the RETURN key is pressed. It then will display your entry and request another.

```
10  REM -- Demonstrate characteristics of string input
20  ! Error 801 is 'Too much data entered'
30  ! Error 803 is 'Illegal character in input'
40  !
100 PRINT 'INPUT A STRING ';                ! Get the string
110 INPUT A$
120 PRINT 'THE STRING IS: '; A$
130 PRINT \ PRINT                           !Skip two lines
140 GOTO 100
```

Running this program gives results as follows:

```
INPUT A STRING? EXAMPLE STRING ONE
THE STRING IS: EXAMPLE STRING ONE

INPUT A STRING? Inserting a, gives an error

    ?Input error 801 at line 110
    ? Placing the ',' in quotes won't help

    ?Input error 801 at line 110
    ? 'So place the entire string with a , in quotes'
    THE STRING IS: So place the entire string with a ,in quotes

    INPUT A STRING? " Will also give an error

    ?Input error 803 at line 110
    ? 'The quotes must match'
    THE STRING IS: The quotes must match

    INPUT A STRING?
```

The example program below requires three inputs, integer, floating point, and string, separated by commas.

```
500 REM -- Enter test parameters
510 PRINT "ENTER TEST NUMBER, NOMINAL VALUE, UNITS";
520 INPUT TX, NV, U$
530 IF TX GOTO 510                    ! Equivalent to IF TX<>0 GOTO 510
```

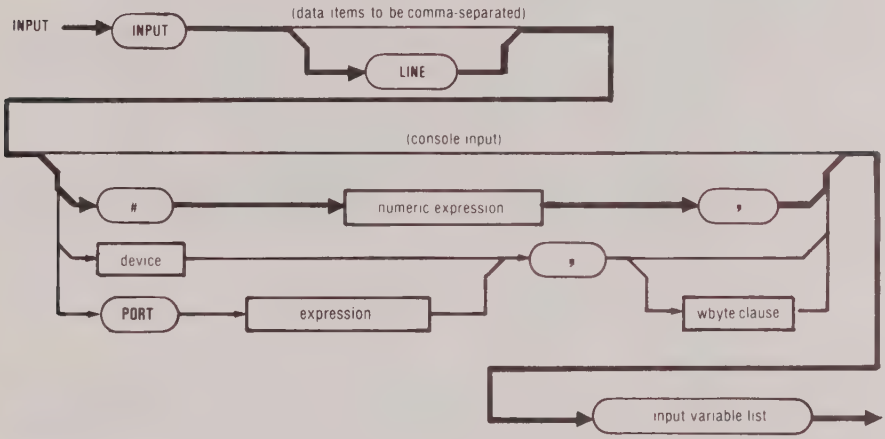
Running this program gives results as follows:

```
ENTER TEST NUMBER, NOMINAL VALUE, UNITS? 101, 2.222, mV
ENTER TEST NUMBER, NOMINAL VALUE, UNITS? 104, 8.888, mV
ENTER TEST NUMBER, NOMINAL VALUE, UNITS? 201, 1,V
ENTER TEST NUMBER, NOMINAL VALUE, UNITS? 203, 100,V
ENTER TEST NUMBER, NOMINAL VALUE, UNITS? 301, 100, mA
ENTER TEST NUMBER, NOMINAL VALUE, UNITS? 302, 1,A
```

Usage

INPUT {# channel}, {variable list}

Syntax Diagram



Description

INPUT is described here for data input from a system level device through a previously opened channel. INPUT may also be used for direct keyboard input, or for input from instruments on the IEEE-488-1978 buses. These applications of INPUT are described individually elsewhere in this manual.

- The optional LINE specification is discussed in the entry for the INPUT LINE #n statement, elsewhere in this section.
- The numeric expression following INPUT or INPUT LINE selects a previously opened channel. Refer to the entry for the OPEN statement.
- The variables that will store the data input are listed, separated by a comma.
- The input data may include integer, floating-point, and string expressions, as well as subranges of arrays and virtual arrays. Refer to Section 6 of this manual set for a discussion of array subranges.

- When the open channel is from a serial (RS-323-C) port, incoming Carriage Return and Line Feed characters are deleted. A Carriage Return, Line Feed sequence is appended after each occurrence of the line terminator character defined by the SET Utility program. If Line Feed or Carriage Return is the line terminator, this process does not duplicate it.

- When the open channel is from a serial (RS-232-C) port, the End-of-File character defined by the SET Utility program is deleted, and CTRL/Z , CHR$(26), is put in its place.

- Input to a two-dimensional array subrange is assigned in the "row-major" manner described in the PRINT discussion in this section. Columns (second dimension) increment most often, and rows (first dimension) increment least often.

- Input to a two-dimensional array subrange must be on a single line separated by commas.

- Input data must be specified in a line of not more than 80 characters when using this method to input values to an array.

- BASIC does not send a prompt character, "?", when input is from a channel.

Examples

In the following example a sequential input channel from the keyboard is opened. When lines 110 and 120 are executed, the message displayed is the string at line 110. This technique allows an INPUT statement, such as line 120, to be used without the usual "?" prompt.

```
10   OPEN "KB0:" AS OLD FILE 2
20   ! Other statements
30   !
110  PRINT "ENTER THE SERIAL NO:   ";
120  INPUT #2, SN$
130  ! Other statements
```

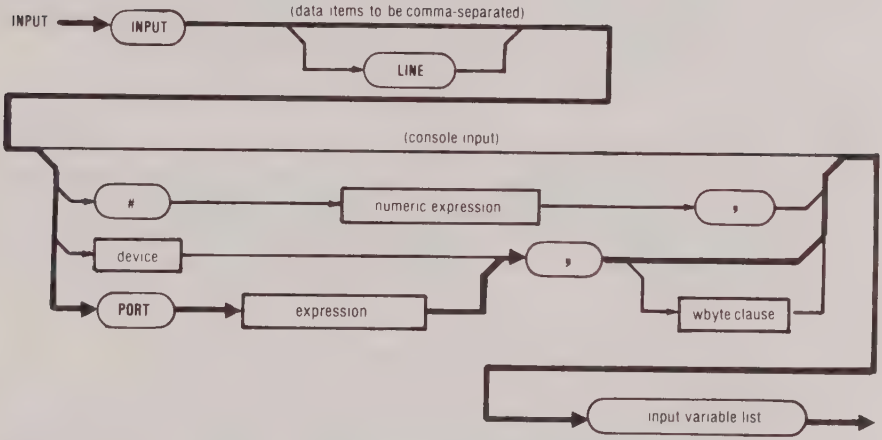
In the following example, assume a device is attached to RS-232-C Port 1 which can print data sent to it and send data to the Instrument Controller. Lines 20 and 30 assign KB1: simultaneously for input and output, using separate channels. This program can send prompts on the output channel (line 100) and receive data on the input channel (line 120). An example of such a device is a printing computer terminal. Note that a "?" is not sent at line 100. This simultaneous assignment for input and output is not possible for a sequential channel to a file.

```
10   REM -- Demonstrate Input and Output From Same Device
20   OPEN 'KB1:' AS OLD FILE 1       ! Input channel 1
30   OPEN 'KB1:' AS NEW FILE 2       ! Output channel 2
40   ! Other statements
50   !
100  PRINT #2, A$                    ! Give prompt
110  INPUT #1, A(0..5)               ! Get 6 values
120  ! Other statements
```

Usage

INPUT{variable}

Syntax Diagram



Description

The INPUT @ statement is used to receive data from instruments on the IEEE-488 Bus.

- The only syntax difference in the INPUT statement for IEEE-488 Bus instruments is the use of a device specification instead of a channel number.
- Only one device number may be specified.
- Input data items must be separated by commas.
- Input data is terminated by a Line Feed character or an alternate character specified by a previous TERM statement. The EOI Bus line may also be used by the transmitting instrument to terminate input.

- The instrument is addressed as a talker prior to reading data.
- The INPUT statement can be structured so that the Instrument Controller will not address an instrument that has been previously addressed. This will increase the speed of data input. INPUT assumes there is a talker already on the bus when the @ character follows INPUT without a device specified. Incoming bus data is then simply assigned directly to the specified variables.

Example

The following example addresses device 2 on port 1 as a talker, and then reads four floating-point (ASCII) values. The first three values should be terminated by commas, and the fourth (last) value should have a Line Feed as a terminator, unless a TERM statement has defined a different terminator character. The EOI line may also be used to terminate the last input.

```
2470 INPUT @ 102, A(1%..4%)
```

following example uses a TERM statement to limit the terminator of the string A$ to the EOI Bus line. This will allow the instrument at device number 10 to transmit binary data or status information.

```
4100    REM -- Get Readings from Instrument
4110    TERM                          ! Limit terminator to EOI
4120    PRINT @ 10, "A3B?"            ! Set up instrument
4130    INPUT @ 10, A$                ! Get data
4140                                  ! other statements
```

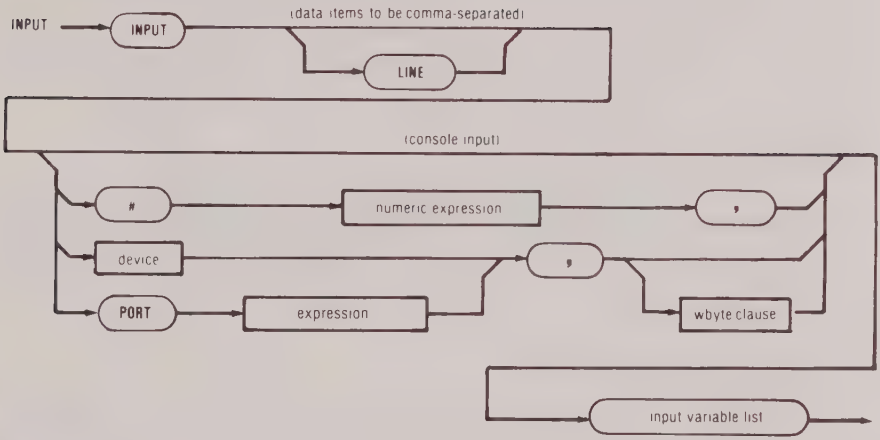
The next example shows how INPUT can be used without a specific device designated.

```
5000    REM -- Get series of readings
5010    INPUT @ 4, A                  ! Get dummy readings
5020    FOR I=1 TO 100
5030    INPUT @, RC                 ! Get 100 readings
5040    NEXT I                        ! from same device
5050                                  ! other statements
```

Usage

INPUT LINE{variable}

Syntax Diagram



Description

The optional LINE specification of the INPUT statement changes the assignment process in the following ways:

- String variables may contain any characters except RETURN, Line Feed, ⟨CTRL⟩/Z, and system control codes (e.g.: ⟨CTRL⟩/P).

- All characters entered, including leading spaces, quotes, and commas, are assigned to the string variable. Only the line terminating character is not assigned.

- The line terminating character is normally RETURN for keyboard input, and Line Feed for the IEEE-488 and RS-232-C ports.

- The line terminating character for both IEEE-488 ports can be redefined in a user program by use of the TERM statement.

- The line terminating character for each RS-232-C port can be individually redefined by SET utility commands that may be included in a command file.

INPUT LINE
Statement

- The line terminating character for keyboard input cannot be redefined.
- Numeric values input from the keyboard may be entered in one line separated by commas, or in multiple lines separated by RETURN entries.
- A question mark prompt will be displayed whenever more data is required.

Examples

The following example requires the operator to enter a string of characters terminated by RETURN, and then enter two numeric values separated by commas or RETURN entries. The string may include quotes and commas. The last numeric entry must be followed by RETURN:

```
INPUT LINE A$, B, C
```

This next example uses a 105 element array to hold 35 sets of three data values each. Line 70 then requests the first five sets of data values:

```
10                                      ! --- Obtain Test Data
20                                      !
30 DIM TD (34,2)                        ! 35 3-element test results
40 PRINT "Enter data for 5 tests:"
50 PRINT \   PRINT
60 PRINT "TEST NUMBER", "NOMINAL VALUE", "TOLERANCE (%)"
70 INPUT LINE TD (0..4,0..2)            ! 5 sets of 3-element data
80                                      !
```

The test data may be entered in many ways. Three examples follow:

```
1. ? 1, 2.2, .02, 2, 4.4, .02,
   ? 4, 8.8, .02, 5, 10, .01

2. ? 1, 2.2, .02
   ? 2, 4.4, .02
   ? 3, 6.6, .02
   ? 4, 8.8, .02
   ? 5, 10, .01

3. ? 1
   ? 2.2
   ? .02
   ? 2
     .
     .
     .
   ? 5
   ? 10
   ? .01
```

The next example uses INPUT LINE to request operator comments for storage in a separate data file. The four blocks reserved will hold up to 2048 characters, or about two full screens of comments. A RETURN entry terminates each line, writes it to the file, and requests the next line. Line 1100 checks first for a RETURN entry with no comments indicating either that the operator has no comments or is done (two RETURN entries).

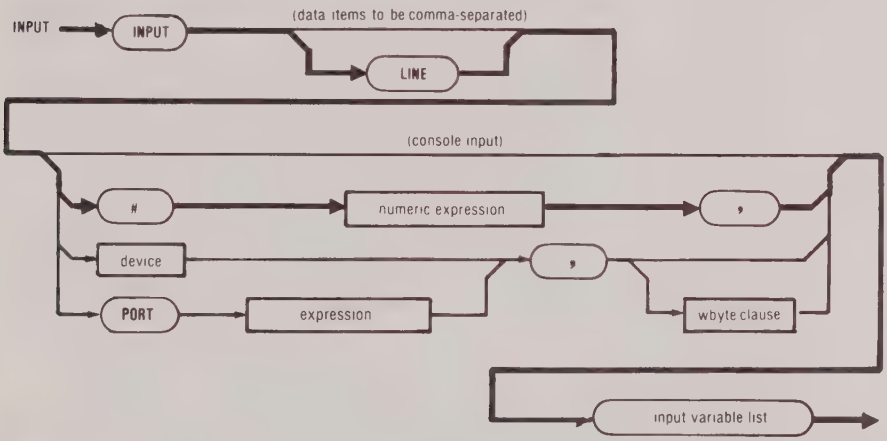
```
1000                  ! Reserve 4 blocks (2048 bytes) for coments:
1010                  !
1020 OPEN "REMARK.DAT" As NEW FILE 3 SIZE 4
1030                  !
1040                  ! Get operator comments on test:
1050                  !
1060 PRINT "Do you have any comments?" \ PRINT
1070 PRINT "Enter RETURN if you have no comments,"
1080 PRINT "or terminate your commments";
1090 PRINT "with two RETURN entries."
1100 INPUT LINE A$
1110 IF A$ <> "" THEN PRINT #3, A$ \ GOTO 1090
1120 CLOSE 3
```

INPUT LINE #n Statement

Usage

INPUT LINE {# channel%}, {variable list}

Syntax Diagram



Description

INPUT LINE #n is described here for data input from a serial device (such as the keyboard or the RS-232 port) through a previously opened channel. The INPUT LINE #n differs from the INPUT #n statement as follows:

- String variables may contain any characters except RETURN, Line Feed, {CTRL}/Z, and system control codes (e.g.: {CTRL}/P).
- All characters entered, including leading spaces, quotes, and commas, are assigned to the string variable. Only the line terminating character is not assigned.
- The input terminating character is Line Feed for the RS-232-C ports.
- The line terminating character for each RS-232-C port can be individually redefined by SET utility commands that may be included in a command file.
- Refer to the INPUT #n statement for additional information.

INPUT LINE #n
Statement

Example

The following example configures the RS-232 port, KB1:, as a bidirectional serial port. Assume the following:

- A RS-232 compatible terminal or teletype-like device is connected to KB1:.
- KB1: has been configured to be compatible with the remote terminal. Use the SET utility program if this is not true.
- The eol (end-of-line) character is: CR, [CHR$(13)].

The example program opens KB1: as channel #1 for input, then opens KB1: as channel #2 for input. The program sends a prompt string to the remote terminal, waits for the required input, then displays the data on the Controller's display.

```
10 CLOSE ALL                          !insurance
20 OPEN "kb1:" AS NEW FILE 2%         !for output to terminal
30 OPEN "kb1:" AS OLD FILE 1%         !for input to controller
40 PRINT #2%, "input data. a, b, and c" !prompt to get data
50 INPUT LINE #1%, A$, B$, C$         ! get it
60 PRINT A$, B$, C$                   ! display on controller
70 GOTO 40
80 END
```

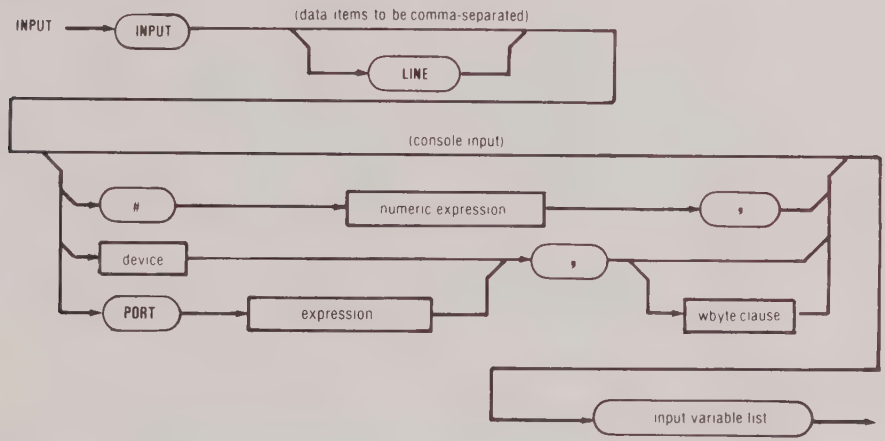
INPUT LINE Statement

IEEE-488

Usage

INPUT LINE @ {device address%, variable}
INPUT LINE {port p%, variable}

Syntax Diagram



Description

The INPUT LINE construction of the INPUT statement allows binary data to be received from the bus and assigned to a string variable.

- The only syntax difference in the INPUT LINE statement for IEEE-488 Bus instruments is the use of a device specification or port number instead of a channel number.
- Only one device number may be specified.
- Input data items may be separated by commas or by the terminating character.
- If PORT P% is used, the named port is addressed directly, without device addressing.
- The terminating character is Line Feed, unless an alternate character is specified by a previous TERM statement.
- The EOI Bus line may also be used by the transmitting instrument to terminate input.

INPUT LINE
Statement

- Each input data item to a string variable may be up to 512 characters in length.
- When a device address is given, the instrument is addressed as a talker prior to reading data.
- Each variable or specified array location must receive an entry.
- The INPUT LINE statement can be structured so that the Instrument Controller will not address an instrument that has been previously addressed. This will increase the speed of data input. INPUT LINE assumes there is a talker already on the bus when the @ character follows INPUT LINE without a device specified. Incoming bus data is then simply assigned directly to the specified variables.
- A string assigned a value by INPUT LINE will include the terminating character.

Examples

The following example allows a less restrictive data format than a simple INPUT statement. Any or all of the ASCII floating point values may be terminated by a Line Feed or the EOI line. Otherwise, this example is similar to the first INPUT statement example (see INPUT).

```
INPUT LINE @ 102, A(1%..4%)
```

LINE is commonly used to read strings which represent numeric data from instruments. Some instruments, however, transmit data with a left justified sign for ease of direct reading on a printer. The resulting string cannot be directly evaluated. The following subroutine removes the spaces between a sign and a numeric string. The string that was created by INPUT LINE is A$.

```
1000  S% = INSTR (1%,A$,'+')          ! Search for a +
1010  IF S% = 0 THEN S% = INSTR (1%,A$,'-') ' If none, search for a -
1020  IF S% THEN S$ = MID(A$, S%, 1%)! S$ is sign, if found
1030  S% = S% + 1%
1040  IF ASCII (RIGHT(A$, S%) = 32% THEN 1030' Skip spaces
1050  A$ = S$ + RIGHT (A$, S%)        ! A$ is now sign followed
1060  REM                             ! by numerics.
```

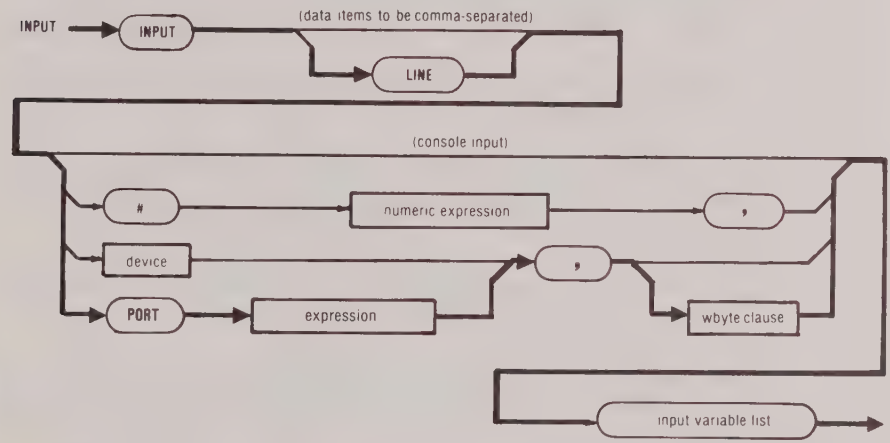
INPUT LINE WBYTE Statement

IEEE-488

Usage

INPUT LINE {Port/device Address,} WBYTE {WBYTE Clause} Variable

Syntax Diagram



Description

The INPUT LINE WBYTE construction of the INPUT statement allows binary data to be received from the bus and assigned to a string variable, and transmits a bus message contained in a WBYTE clause prior to receiving each data item.

- Only one device number may be specified.
- Input data items may be separated by commas or by the terminating character.
- The terminating character is Line Feed, unless an alternate character is specified by a previous TERM statement.
- The EOI bus line may also be used by the transmitting instrument to terminate input.
- Each data item input to a string variable may be up to 80 characters in length.

INPUT LINE WBYTE
Statement

- The instrument is addressed as a talker only once, prior to all data readings.

- INPUT LINE WBYTE can be structured so that the Instrument Controller will not address an instrument that has been previously addressed. This will increase the speed of data input. INPUT LINE WBYTE assumes there is a talker already on the bus when the @ character follows INPUT LINE without a device specified.

- A string assigned a value by INPUT LINE WBYTE will include the terminating character.

- The WBYTE clause selects a subrange of an integer array for output. This may contain addressing, bus messages and device-dependent data. See the discussion of the WBYTE statement.

- The WBYTE clause does not need to be sent to the same instrument port from which input is being taken.

- Make sure that the WBYTE output does not unaddress the instrument as a talker if it is sent to the port from which input is read.

- After the WBYTE output, the specified instrument is addressed as a talker and one data line is read.

- The WBYTE clause is then sent again and the next line of input is read. This process is repeated until the input data list has been satisfied.

Examples

The following example sends a one-character trigger to an already-addressed listener on port 1 via the WBYTE clause. It then addresses device 1 on port 0 as a talker. Each floating-point ASCII value read from port 0 is placed in an element of A after the trigger command is sent.

The sequence is:

1. Send A%(0%) on port 1
2. Read A(0%) on port 0
3. Send A%(0%) on port 1
4. Read A(1%) on port 0

. . .

. . .

39. Send A%(0%) on port 1
40. Read A(19%) on port 0

```
3750 INPUT LINE @ 1%, {WBYTE PORT 1%, A%(0%)} A(0%..19%)
```

The following example initializes instrument port 0 and then sends a string of bus messages via the WBYTE clause. The array A% contains the following six commands: A%(0) = UNL, A%(1) = UNT, A%(2) = MLA 2, A%(3) = GET, A%(4) = UNL, and A%(5) = MTA 2. Instrument device 2 on port 0 is triggered 50 times, each time reading a string into the next element of array A$.

INPUT LINE WBYTE
Statement

The sequence is:

1. Send A%(0%..5%)
2. Read A$(0%)
3. Send A%(0%..5%)
4. Read A$(1%)

. . .

. . .

99. Send A%(0%..5%)
100. Read A$(49%)

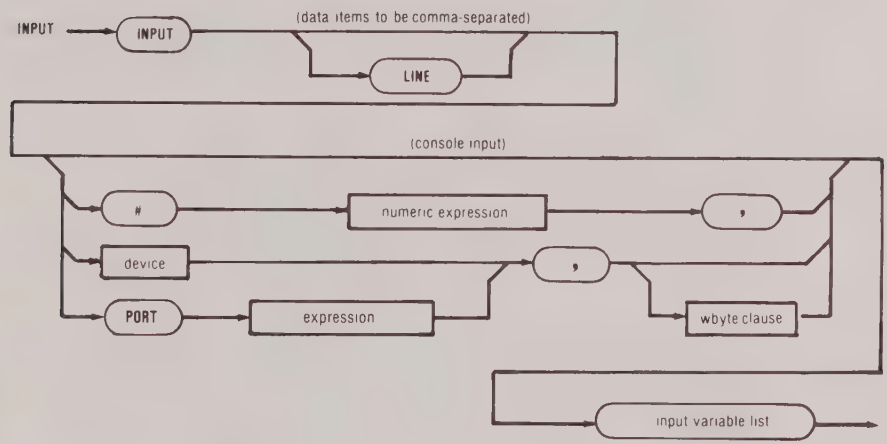
```
10  INIT PORT 0%
20  INPUT LINE PORT 0,{WBYTE A%(0%..5%)} A$(0%..49%)
```

IEEE-488

Usage

INPUT WBYTE @{device variable}
INPUT WBYTE{port p% variable}

Syntax Diagram



Description

INPUT WBYTE transmits a Bus message contained in a WBYTE clause prior to receiving each data item.

- Only one device number may be specified.
- Input data items must be separated by commas.
- The terminating character is Line Feed, unless an alternate character is specified by a previous TERM statement.
- The EOI Bus line may also be used by the transmitting instrument to terminate input.
- The instrument is addressed as a talker only once, prior to all data readings.

INPUT WBYTE
Statement

- INPUT WBYTE can be structured so that the Instrument Controller will not address an instrument that has been previously addressed. This will increase the speed of data input. INPUT WBYTE assumes there is a talker already on the bus when the @ character follows INPUT without a device specified.

- The WBYTE clause selects a subrange of an integer array for output. This may contain addressing, bus messages and device dependent data. See the discussion of the WBYTE statement in this section.

- The WBYTE clause does not need to be sent to the same instrument port from which input is being taken.

- Make sure that the WBYTE output does not unaddress the instrument as a talker if it is sent to the port from which input is read.

- The WBYTE clause is then sent again and the next input data item is read. This process is repeated until the input data list has been satisfied.

- After the WBYTE output, the specified instrument is addressed as a talker and one data item is read.

Examples

The following example sends a one-character trigger to an already-addressed listener on port 1 via the WBYTE clause. It then addresses device 1 on port 0 as a talker. Each floating-point ASCII value read from port 0 is placed in an element of A after the trigger command is sent.

The sequence is:

1. Send A%(0%) on port 1
2. Read A(0%) on port 0
3. Send A%(0%) on port 1
4. Read A(1%) on port 0

. . .

. . .

39. Send A%(0%) on port 1
40. Read A(19%) on port 0

```
3750    INPUT @ 1%, (WBYTE PORT 1%, A%(0%)) A(0%..19%)
```

INPUT WBYTE
Statement

The following example initializes instrument port 0 and then sends a string of bus messages via the WBYTE clause. The array A% contains the following six commands: A%(0) = UNL, A%(1) = UNT, A%(2) = MLA 2, A%(3) = GET, A%(4) = UNL, and A%(5) = MTA 2. Instrument device 2 on port 0 is triggered 50 times, each time reading a string into the next element of array A$.

The sequence is:

1. Send A%(0%..5%)
2. Read A$(0%)
3. Send A%(0%..5%)
4. Read A$(1%)

. . .

. . .

99. Send A%(0%..5%)

100. Read A$(49%)

```
10      INIT PORT 0%
20      INPUT PORT0, (WBYTE A%(0%..5%)) A$(0%..49%)
```

INSTR() Function

Format

INSTR (start%, string$, substring$)

Description

The INSTR function searches for a specified substring within a string and returns the starting location of the substring.

- INSTR has an integer (start%) and two strings (string$ and substring$) as arguments, and returns an integer.
- The integer argument is the starting character position for the search.
- The first string is the string to be searched.
- The second string is the substring to be searched for.
- The substring must be exactly and wholly contained within the string for the search to be successful.
- INSTR returns 0 when the substring is not found.
- INSTR returns 0 when the starting character position is greater than the length of the string.
- INSTR returns, in integer form, the position in the string of the first character of the substring when the substring is found.
- INSTR returns the number of the starting character position when the substring is null, and the starting character position is less than the length of the string.

Example

The following program generates the interaction printed below it.

```
1000  ! Demonstrate INSTR Function
1020  A$ = This string has 35 characters in it"
1030  PRINT "The given string is: "; A$
1040  PRINT "Type the substring to be found";
1050  INPUT B$ \ PRINT
1060  PRINT "Type the start position ";
1070  INPUT S% \ PRINT
1080  PRINT "INSTR ("; S%; ', "'; A$; '", "'; B$; '") =' ;
1100  PRINT INSTR (S%, A$, B$) \ PRINT
1120  GOTO 1040
```

INSTR()
Function

The results of running this program are:

```
The given string is: This string has 35 characters in it
Type the substring to be found? has
Type the starting position? 6
INSTR (6, "This string has 35 characters in it", "has") = 13
```

Note that the INSTR function returned 13 -- the "h" of "has".

```
Type the substring to be found? has
Type the starting position? 14
INSTR (14, "This string has 35 characters in it", "has") = 0
```

There is no "has" after character 14 to the end of A$, so the INSTR function returned 0.

```
Type the substring to be found? strung
Type the starting position? 1
INSTR (1, "This string has 35 characters in it", "strung") = 0
```

There is no "strung" in A$, so the INSTR function returned 0.

```
Type the substring to be found? IN
Type the starting position? 1
INSTR (1, "This string has 35 characters in it", "IN") = 0
```

Note that the search is sensitive to whether or not the letters are capitalized.

```
Type the substring to be found? in (space)
Type the starting position? 1
INSTR (1, "This string has 35 characters in it", "in") = 9
```

"in" is found embedded in the word "string". It begins at character 9 of A$.

```
Type the substring to be found? in
Type the starting position? 1
INSTR (1, "This string has 35 characters in it", "in ") = 31
```

Search for the three letters i - n - (space) to find the word "in" by itself, rather than "in" in "string".

```
Type the substring to be found?
Type the starting position? 5
INSTR (5, "This string has 35 characters in it", "") = 5
```

A null substring returns the starting location.

```
Type the substring to be found?
Type the starting position? 36
INSTR (36, "This string has 35 characters in it", "") = 0
```

A search for a null string beyond the end of A$ returns 0.

INT() Function

Format

INT (numeric expression)

Description

The INT function returns the largest integer value that is less than or equal to the specified floating-point or integer number.

- INT has a floating-point input number and returns a floating-point result. The result may be assigned to an integer variable.
- The domain of input values is any positive or negative floating-point value, or zero.
- The range of output values is positive and negative floating-point values, and zero.
- If INT is given an integer argument, e.g., INT (A%), INT will return the integer argument unchanged (i.e. INT(A%) = A%).

Example

These immediate mode examples show how the INT function works:

```
PRINT INT(12.34)
12
PRINT INT(0.097346)
0
```

An integer value is returned unchanged:

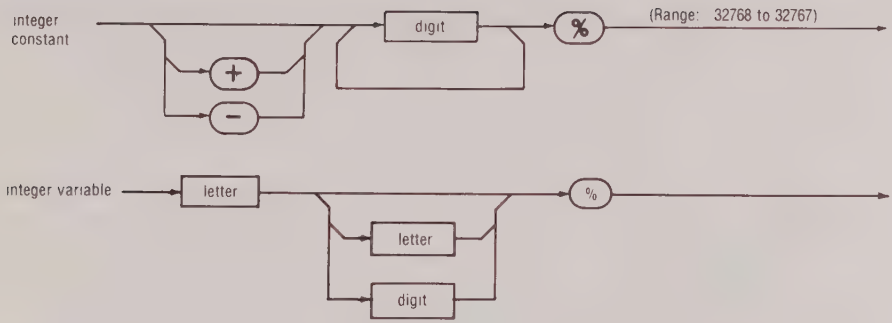
```
PRINT INT(-2)
-2
```

Negative arguments produce:

```
PRINT INT(-7.009)
-8
```

Note that the INT function returns the largest integer value that is less than or equal to the entire input value. For negative numbers, this will be one smaller than the integer portion of the input number (-8 in the example).

Syntax Diagram



Integer data has the following characteristics:

- Range: –32768 to +32767
- Resolution: Integers
- Exactness
- Memory requirement (per datum): 2 bytes
- Integer data is represented internally in binary but displayed by the Instrument Controller in decimal without the modification process described in Floating-Point Data.
- Operations that call for an integer result are rounded to an integer, if necessary.

INTEGER Data

Integer Constants

Integer constants are whole numbers identified by a "%" suffix on the number.

Examples

-0%

5%

-32000%

-40000% Outside the allowed range.

Integer Variables

Integer variables are designated by a floating-point variable name followed by a "%" character.

System Variable

Description

The system variable KEY contains the number of the last Touch-Sensitive Key that was pressed. The KEY variable has the following characteristics:

- KEY is an integer ranging from 0 to 60.
- A KEY value of 0 means that no Touch-Sensitive Key has been pressed since the last time the value of KEY was used by a BASIC statement or since the last time a RUN command was executed.
- When KEY is used by a program (e.g., "K% = KEY" or "IF KEY = 0 THEN..."), it is subsequently set to 0.
- KEY may be used in any context that requires an integer variable.
- KEY cannot be assigned a value except by pressing the Touch-Sensitive Display.

Example

The following example shows the KEY variable being used to determine Touch-Sensitive Keyboard entries.

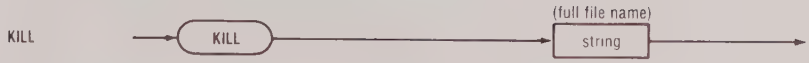
```
1200 K% = KEY
1201 IF K% <> 0 THEN 2600      ! If the display's been touched
1202                           !     goto the the key handler
.
.
.
2600 ! Key handler
2610 ! Look for keys 21 or 41
2620 IF K% = 21 GOTO 3000          ! Go to YES if 21 was touched
2630 IF K% = 41 GOTO 4000          ! Go to EXIT if 41 was touched
2640 PRINT CHR$(7);                ! Beep and
2650 GOTO 1200                     ! repeat if any others were touched
```

KILL Statement

Usage

KILL "filename"

Syntax Diagram



Description

The KILL statement deletes a specified file.

- A file name must be specified, enclosed in single or double quotes.
- The default file name extension is ". " (3 spaces). Any other file name extension must be specifically stated.
- Error 305 results if an attempt is made to kill a file that is not found.
- The file is deleted from the default System Device (SY0:) if the file name is not preceded by a device sepcification.

Example

The following example illustrates the KILL statement used to delete FILE.DAT from the directory of the System Device.

```
KILL "FILE.DAT"
```

This next example has a file of four blocks (2048 characters) to save intermediate status information during the program. This "check pointing" file may be used to recover from power failures, etc. When the program terminates, the file is no longer needed, so it is deleted at line 32766.

```
10     REM -- Meter Calibration Program
20     OPEN "CHECK.PT" AS NEW FILE 2 SIZE 4  ! Check pointing file
       .
       .
       .
32766  KILL "CHECK.PT"                      ! Delete check pt file
32767  END
```

Format

LCASE$(argument string$)

Description

The LCASE$ function converts a string to lower case.

- The argument must be a string variable.
- The output is a string (of the same length as the argument string$) with all alphabetic characters converted to lower-case.

Example

This example shows a string which is received from the keyboard being converted to lower case before decoding.

```
120 INPUT A$                        ! Fetch a string from the user
130 AL$=LCASE(A$)                   ! and convert it to lower case
131                                 ! before use.
140 IF ASCII$(AL$)>96 AND ASCII$(AL$)<101 THEN 1000 ELSE 120
141                                 ! Look for a-d only
```

Usage

LEFT (string$, number of characters%)

Description

The LEFT function returns a substring of the specified string, starting from the left. The number of characters returned is, if possible, the number specified by the second argument.

- LEFT has a string and an integer as arguments, and returns a string.
- LEFT returns a null string when the number of characters is specified as 0.
- LEFT returns an identical string when the number of characters is specified equal to or longer than the string.

Example

In the following examples, the string A$ contains the characters "THIS IS THE FIRST EXAMPLE STRING", and L% = 11%.

STATEMENT	RESULT
X$ = LEFT (A$, L%)	X$ = "THIS IS THE", the leftmost 11 characters of A$.
PRINT X$; " LEFT PART"	Displays "THIS IS THE LEFT PART".

LEN() Function

Format

LEN (string$)

Description

The LEN function returns the number of characters contained in a string.

- LEN has a string as an argument, and yields an integer.
- The string count includes leading and trailing blanks and null characters.
- LEN returns 0 from a null string (no characters) input.

Example

The following examples illustrate the results of uses of the LEN function.

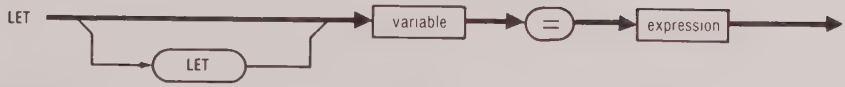
STATEMENT	RESULTS
X% = LEN (A$)	Place the length of string A$ into the integer variable X%.
PRINT LEN (A$)	Display the length of A$. For example, if A$ contains "HELLO!", 6 is displayed.

Statement

Format

[LET] {variable} = {expression}

Syntax Diagram



Description

The LET statement evaluates an expression and assigns the results to a specified variable.

- The data types of the expression and the variable must be either numeric or string, not mixed.
- A numeric expression will be converted if necessary to integer or floating point as required by the variable.
- The word LET is optional.
- The = sign is an assignment operator. It does not imply equality.
- The absolute value of a floating-point value is rounded to an integer, the proper sign is assigned, and the value is then assigned to the integer variable.

LET
Statement

Example

The following examples show the results of assigning values of expressions of one type to variables of another type:

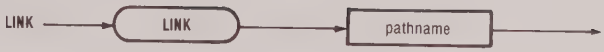
Statement	Result
LET A%=-3.2	A% has value -3
B%=-12.6	B% has value -13
LET A1%=4.8	A1% has value 5
X=12%+A1%	X has value 17
Y="12E3"	Error 600, no change in Y
S$="HELLO"+"THERE"	S$ contains HELLO THERE
LET SX$=12%	Error 600, no change in SX$
N% = N% + 1%	N% is incremented by 1

LINK Statement

Usage

LINK{filename}

Syntax Diagram



Description

The LINK statement loads an object file which contains one or more Assembly Language or FORTRAN subroutine(s) into the Instrument Controller memory.

- The object file is named in the pathname shown in the LINK syntax diagram. The pathname must contain the file name and may also contain a device name and a file extension. The pathname must be enclosed in quote characters (").

  1. If a device name is not specified, the System Device (SY0:) is searched for the file.

  2. If a file extension is not specified, the Instrument Controller uses the default extension .OBJ.

- If the specified file is not found, error code 305 (file not found) is displayed.

- If the specified file is found, the subroutine(s) in the file is(are) loaded into the Instrument Controller memory.

- Multiple LINK commands can be used to load different object files.

- LINK can be used in both Immediate and Run modes.

- LINK commands must follow all COM statements in the program because COM storage must be defined before loading any subroutine(s).

Example

```
14225 LINK "tests.rep"          ! Load the subroutines in the
14226                           ! "tests.rep" object file
```

Immediate Mode Command

Format

LIST [linenumber]
LIST [start-finish linenumbers]

Description

The LIST command displays a program or a portion of a program in line number order.

- Display starts at the first line of a program and proceeds to the last line if line numbers are not specified.
- One line is displayed if a single line number is specified. The command is ignored if the line does not exist.
- A portion of a program is displayed if two line numbers are specified. The display will be the lines with numbers between and including the specified lines numbers if they exist.
- If the portion to be listed is larger than one display page (16 lines), the display will scroll upwards until the last line specified has been displayed.
- Use Page Mode and the NEXT PAGE key, or ⟨CTRL⟩/S and ⟨CTRL⟩/Q to stop and restart the display. These functions are discussed earlier in Section 4 of this manual set.

Examples

The following examples illustrate common uses of the LIST command:

COMMAND	RESULTS
`LIST`	Displays the entire program in memory from the first line.
`LIST 500`	Displays only line 500 of the program, if it exists.
`LIST 600-800`	Displays a program segment beginning with the first line after 599 and ending with the last line before 801.

Format

LN (numeric expression)

Description

The LN function returns a floating-point number equal to the natural logarithm of the input value.

- LN has a floating-point argument and returns a floating-point result.
- The natural logarithm is the exponential power that, if applied to the number e, produces the given input value.
- The value used by the Instrument Controller for e is 2.71828182845905
- The domain of input values is positive numbers. Error 606 results when zero or a negative number is used as input.

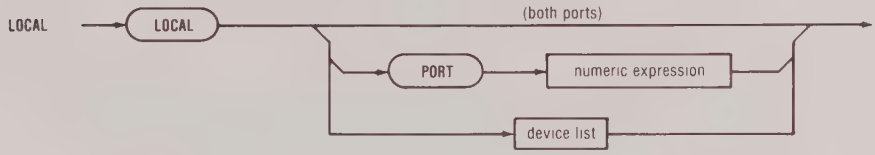
Example

```
PRINT LN(5014)
8.519989
```

Usage

LOCAL [device list or port expression]

Syntax Diagram



Description

LOCAL resets instruments to a local state. Typically, this means that Front Panel controls are activated.

Without a device list, LOCAL is the reverse of REMOTE.

- REN is set false on both ports if a port number is not specified.
- REN is set false only on the designated instrument port when a port is specified.
- LOCAL, without a device list, reverses all effects of the LOCKOUT statement.

With a device list, LOCAL sends a GTL (Go To Local) message on the ports identified in the device list.

- The instruments specified are first addressed as listeners.
- The GTL message is then sent.
- If LOCKOUT was previously sent, the GTL message will not activate the front panel of the instrument.

LOCAL
Statement

Example

The following examples illustrate common usage of the LOCAL statement:

STATEMENT	RESULT
LOCAL	Set REN false on both ports.
LOCAL PORT 1	Set REN false on port 1.
LOCAL @ 103:4	Send GTL message to the instrument on port 1 at address 3, secondary address 4.
LOCAL @ 7 @ 113	Send GTL messages to the instrument on port 0 at address 7, and to the instrument on port 1 at address 13.

The following example illustrates the use of LOCAL to return an instrument temporarily to local control. Line 1180 returns the instrument to REMOTE control.

```
110     INIT                            ! Initialize both ports
120     REMOTE @ 2 @ 104                ! Place instruments in remote
130                                     ! other statements
140                                     !
1100    REM -- Subroutine: Special Instrument Usage
1110    LOCAL @ 104                     ! Set one instrument to local
1120    PRINT "SET INSTRUMENT TO PERFORM TEST"
1130                                    ! other statements
1140                                    !
1180    REMOTE @ 104                    ! Place instrument in remote
```

Usage

LOCKOUT [PORT p%]

Syntax Diagram

Description

The LOCKOUT statement disables not only Front Panel controls (as with REMOTE) but also any "return to local" function button that may be on an instrument. As defined in the IEEE-488-1978 Standard, any instrument addressed to listen after receiving a local lockout command will immediately be placed in the "Remote With Lockout State".

- LOCKOUT implements the LLO (Local Lockout) capability on the IEEE-488 Bus.
- LOCKOUT sets REN, and then sends LLO.
- This sequence is sent on both ports if no port number is given.
- This sequence is sent on only one port if a port number is specified.

Example

The following example uses the REMOTE statement with a device list after the LOCKOUT statement to immediately disable all local instrument control. The device addresses sent at line 160 place the instruments in "Remote With Lockout State" since LOCKOUT was previously sent at line 150.

```
100     REM--Initialize Ports, Disable Local Instrument Control
110     ! Instruments are at Port 0: address 2, 7, 9
120     !                            And at Port 1: address 1
130     !
140     INIT
150     LOCKOUT
160     REMOTE @ 2 @ 7 @ 9 @ 101
170     ! other statements
```

LOCKOUT
Statement

The following example uses LOCKOUT to place instruments on port 0 into a state such that whenever they are used, they enter the "Remote With Lockout State". The CLEAR statement at line 100 insures that no previously addressed instrument is placed in "Remote With Lockout State". At line 140, when the instrument at address 2 of port 0 is addressed as a listener and sent the message "F2R0", it is placed in "Remote With Lockout State" (as previously required by line 100).

```
30       ! other statements
40       !
100      CLEAR                          ! Unlisten, Untalk all devices
110      LOCKOUT PORT 0
120      ! other statements
130      !
140      PRINT @ 2, "F2R0"
```

Format

LOG (numeric expression)

Description

The LOG function returns a floating-point number equal to the common logarithm (log base 10) of the input value.

- LOG has a floating-point argument and returns a floating-point number.
- The common logarithm is the exponential power that, if applied to the number 10, produces the given input value.
- The domain of input values is positive numbers. Error 606 results when zero or a negative number is used as input.
- The range of output values is –307 to +308.

Example

```
PRINT LOG(5014)
3.700184
```

Description

Logical operators AND, OR, XOR, NOT, operate on the binary digits that make up an integer. NOT is a unary operator, acting upon one integer. AND, OR, and XOR are binary operators, using the bits of one integer to act upon another integer. Logical operators allow examination or modification of integer bit patterns when they have been used to store binary data, such as status or binary readings from instrumentation.

Binary Numbers

To use logical operators effectively, it is necessary to understand the binary number system and how the Instrument Controller uses binary numbers to represent integers.

The Instrument Controller uses a 16-bit (2-byte) word to store each integer. Each bit position represents a weighted power of 2. The sum of the weight column in the following chart is the number of separate integers that can be represented with 16 bits less one because zero is an additional integer.

BIT POSITION	WEIGHT
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128
8	256
9	512
10	1024
11	2048
12	4096
13	8192
14	16384
15	32768
SUM:	65535

Integer numbers are represented by setting appropriate bit positions to 1. Adding the weighted values of each position that is set to 1 gives the decimal value of the integer. For example, the number 305 is represented by the binary pattern 0000 0001 0011 0001, as follows:

Position: 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

Setting: 0 0 0 0 0 0 0 1 0 0 1 1 0 0 0 1

Since each bit that is set to 1 carries a binary weight, it is possible to verify that the binary pattern is correct. Adding the numbers 256 + 32 + 16 + 1 gives the decimal value 305.

Decimal numbers can be converted to binary by continuously dividing by 2 and keeping track of the remainders (always 1 or 0). For example, the number 305 is converted to binary as follows:

305 / 2 = 152 R = 1
152 / 2 = 76 R = 0
76 / 2 = 38 R = 0
38 / 2 = 19 R = 0
19 / 2 = 9 R = 1
9 / 2 = 4 R = 1
4 / 2 = 2 R = 0
2 / 2 = 1 R = 0
1 / 2 = 0 R = 1

Reading the remainder bits, from the bottom up, gives the result 100110001, the binary representation of 305.

Twos Complement Binary Numbers

By using the most significant bit (15) to identify a negative integer, the Instrument Controller divides the pattern into 32767 positive numbers, the number 0, and 32768 negative numbers. Negative numbers are represented in a form called twos complement. To change either to or from twos complement form, the following steps are required:

1. Replace every 1 with a 0.
2. Replace every 0 with a 1.
3. Add 1.

For example, to change the pattern 0000 0001 0011 0001 (+305) to twos complement form, first reverse 1's and 0's: 1111 1110 1100 1110, and then add 1: 1111 1110 1100 1111 (–305). To change it back, first reverse 1's and 0's: 0000 0001 0011 0000, and then add 1: 0000 0001 0011 0001.

AND Operator

AND returns an integer bit pattern with a 1-bit in every position where both of two input integers have a 1-bit. The AND operator is useful to check for the setting of particular bit(s) to 1 by operating on an unknown status word with a mask word (number) having the appropriate bit(s) set to 1. The following examples illustrate the results of AND operations:

33% AND 305%	33% 0000 0000 0010 0001 305% 0000 0001 0011 0001
RESULT: 33%	0000 0000 0010 0001
–74% AND 305%	–74% 1111 1111 1011 0110 305% 0000 0001 0011 0001
RESULT: 304%	0000 0001 0011 0000

OR Operator

OR returns an integer bit pattern with a 1-bit in every position where either of two input integers have a 1-bit. The OR operator can be used to check for the resetting of particular bit(s) to 0 by operating on an unknown status word with a mask word (number) having the appropriate bit(s) set to 0. The following examples illustrate the results of OR operations:

33% OR 305%	33% 0000 0000 0010 0001 305% 0000 0001 0011 0001
RESULT: 305%	0000 0001 0011 0001

-74% OR 305%	-74%	1111 1111 1011 0110
	305%	0000 0001 0011 0001
RESULT: -73%		1111 1111 1011 0111

XOR Operator

XOR (Exclusive OR) returns an integer bit pattern with a 1-bit in every position where the bits of two input integers are opposite. A mask word applied to an unknown integer through XOR will invert (1 to 0, and 0 to 1) all bit positions where the mask contains a 1, and leave unchanged all bit positions where the mask contains a 0. The following examples illustrate the results of XOR operations:

33% XOR -1%	33%	0000 0000 0010 0001
	-1%	1111 1111 1111 1111
RESULT: -34%		1111 1111 1101 1110
-74% XOR 0%	-74%	1111 1111 1011 0110
	0%	0000 0000 0000 0000
RESULT: -74%		1111 1111 1011 0110

NOT Operator

NOT is a unary operator that operates upon a single integer. NOT returns a 1 bit in every position where the input integer had a 0 bit, and a 0 bit in every position where the input integer had a 1 bit. The following examples illustrate the results of NOT operations:

NOT 33%	33%	0000 0000 0010 0001
RESULT: -34%		1111 1111 1101 1110
NOT -74%	-74%	1111 1111 1011 0110
RESULT: 73%		0000 0000 0100 1001

LSH() Function

Format

LSH(arg%,count%)

Description

The LSH() function performs an unsigned logical shift on a binary integer (arg%) by shifting it by a given number of bits (count%).

- The function arguments are considered to be integers; any floating-point arguments will be truncated to integer.
- This operation may cause an arithmetic overflow (error 601) to be reported if the argument cannot be represented as an integer.
- The LSH function shifts arg% right or left by count% bits. This is a logical (or unsigned) shift, which means that, unlike the ASH function, zeroes are propagated (shifted in from the left) when a right shift is performed.

Examples

Right Shifts

A right shift is accomplished by using one of the two shift functions, ASH() or LSH(), with a negative shift count. The shift is performed by taking the absolute value of the shift count and then shifting the argument right by that number of bits.

The LSH() (logical shift) function does not copy the sign bit; it unconditionally copies zeroes from the left. As an example:

```
100 print "Shift count", "Hex", "Decimal"
110 for i%=0% to -15% step -1%
120   j% = lsh(-32768%, i%)
130   print i%, rad$(j%, 16%), j%
140 next i%
```

LSH()
Function

prints the following:

```
Shift Count     Hex      Decimal
 0              8000     -32768
-1              4000      16384
-2              2000       8192
-3              1000       4096
-4               800       2048
-5               400       1024
-6               200        512
-7               100        256
-8                80        128
-9                40         64
-10               20         32
-11               10         16
-12                8          8
-13                4          4
-14                2          2
-15                1          1
```

NOTE

The sign bit is not copied as the number is shifted to the right, so that when a negative number is right-shifted it becomes a positive number.

Left Shifts

A left shift is accomplished by using one of the two shift functions, ASH() or LSH(), with a positive shift count. The shift is performed by shifting the argument left by the number of bits indicated.

In the case of a left shift both ASH() and LSH() generate identical results. The program:

```
100 print "Shift count", "Hex", "Decimal"
110 for i%=0% to 15%
120    j% = lsh(1%, i%)
130    h$ = rad$(j%, 16%)
140    print i%, dupl$("0", 4% - len(h$)); h$, j%
150 next i%
```

prints the following:

Shift Count	Hex	Decimal
0	0001	1
1	0002	2
2	0004	4
3	0008	8
4	0010	16
5	0020	32
6	0040	64
7	0080	128
8	0100	256
9	0200	512
10	0400	1024
11	0800	2048
12	1000	4096
13	2000	8192
14	4000	16384
15	8000	-32768

Remarks

Compare the LSH() function to the ASH() function.

Description

The mathematical functions supplied with Fluke BASIC include an assortment of tools to simplify computation tasks on collected data. Included are a square root function, natural and common logarithms, exponentiation of e, absolute value, sign, and greatest integer.

- Mathematical functions operate on integer or floating-point numbers.
- Mathematical functions accept as input an expression that evaluates to a numeric quantity.
- Mathematical functions return an integer or floating-point number.
- Conversion between numeric data types is automatic where required by the function or specified by the user. This conversion process requires additional processing time.
- The domain of acceptable input values for some functions is limited or not continuous.
- The range of resulting output values for some functions is not continuous or has points of underflow (too close to zero) or overflow (too large).
- The definition of each function includes limitations of domain and range.

Refer to the individual descriptions of each math function for details:

SQR	square root
LN	natural logarithms
LOG	common logarithms
EXP	exponentiation of e
ABS	absolute value
SGN	signum
INT	greatest integer
MOD	remaindering

System Variable

Description

The MEM System Variable contains the amount of user memory that is currently available expressed in Bytes.

- MEM contains a floating-point value.
- MEM may be used in any context that requires a floating-point variable.
- MEM cannot be assigned a value by a program.

Example

This example displays the contents of the MEM variable.

```
PRINT MEM      ! Display the amount of memory available
```

The example below shows a test to determine if there is less than 1024 bytes of free memory.

```
IF MEM < 1024 THEN 2400  ! Check for less than 1K available
```

Format

MID (string$, start%, number of characters%)

Description

The MID function returns a substring of the specified string, starting from the specified character position, and including the specified number of characters.

- MID has a string and two integers as arguments, and returns a string result.
- MID(A$, S%, N%) is equivalent to LEFT(RIGHT(A$, S%), N%).
- MID returns a null string when the number of characters specified is zero.
- MID returns a null string when the starting character position specified is more than the length of the string.
- MID returns an identical string when the starting character position specified is 0 or 1, and the number of characters specified is equal to or longer than the string.
- Arguments entered as real numbers will be truncated to the integer value.

Example

In the following examples, the string A$ contains the characters "THIS IS THE FIRST EXAMPLE STRING".

STATEMENT	RESULT
Z$ = MID (A$, 13%, 13%)	Z$ = "FIRST EXAMPLE", 13 characters of A$ starting at position 13.
X$ = MID (A$, 2.6, 50)	Z$ = "HIS IS THE FIRST EXAMPLE STRING". Note that 2.6 was truncated to 2 and that length = 50 returned the remainder of the string.

Format

MOD(expression, expression)

Description

The MOD() (modulo) function returns the remainder that is produced by dividing two integer or floating-point numbers. The MOD() function is defined as

$$\text{mod}(x, y) = x - (y * \text{truncate}(x / y))$$

in which the truncate operation simply drops any fractional result produced by the division of x by y. The result is x treated as a number modulo y; this result always has the same sign as x. The result type returned by MOD() is integer only if both arguments are integers; if either or both arguments are floating point the computation will be performed using floating-point arithmetic.

Example

```
PRINT MOD(11,3)
```

The number 3 will divide into 11 three times (9) leaving a remainder of two.

```
PRINT MOD(2,9)
```

Two doesn't divide into nine at all -- the remainder is two.

The program below uses the MOD function to help convert numbers from decimal to octal. This can also be done with the RAD$() function.

```
11055 ! Convert Decimal (1-63) to Octal (1-77)
11060 PRINT "Enter decimal";
11070 INPUT D                                  ! Fetch decimal
11075 IF D > 63 OR D < 1 GOTO 11060            ! Dump any out of range
11080 W=INT(D/8)*10                            ! Compute whole part
11090 R=MOD(8,D)                               ! Compute remainder
11100 PRINT "Octal equivalent is ";W+R         ! Display the conversion
11110 GOTO 11060
```

Format

NUM$ (integer or floating point number)
NUM$ (integer or floating point number, string)

Description

The NUM$ function returns a string of characters in the format that a PRINT or PRINT USING statement would output the given number.

- NUM$ has an integer or a floating-point number and an optional character string as arguments, and yields a numeric character string with appropriate spaces and decimal.

- When input is limited to an integer or floating-point number, NUM$ returns a character string in the format that PRINT would output the number. For example, if the input number is 1.00000, NUM$ returns " 1 ". (Note leading and trailing space.)

- When input includes a comma followed by a character string, NUM$ returns a character string in the same format that PRINT USING would output the number using the specified string. For example, if the input is 5.5, with the format string "#.##^^^", NUM$ returns "5.50E+00".

- Refer to the discussion of the PRINT USING statement for proper formatting of the input string and the resulting output string.

Example

The following examples illustrate the results of using the NUM$ function.

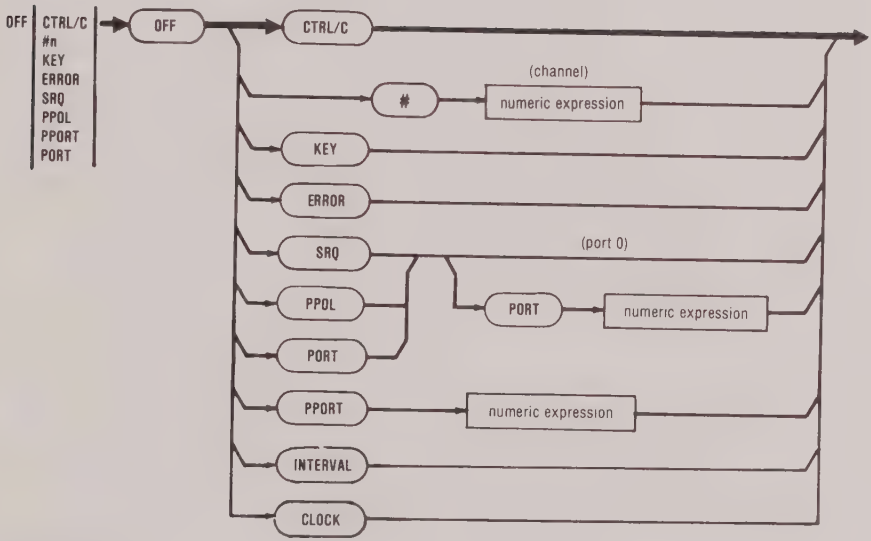
STATEMENT	RESULT
Y$ = NUM$ (10.5)	Y$ is assigned the string " 10.5 ".
Y$ = NUM$ (X * A)	Same as above, except that an expression is used.
Z$ = NUM$ (10.5, "##.##")	Z$ is assigned the string "10.50".
Z$ = NUM$ (X, S$)	Same as above, except a format, S$, is used.

OFF CTRL/C Statement

Usage

OFF CTRL/C

Syntax Diagram



Description

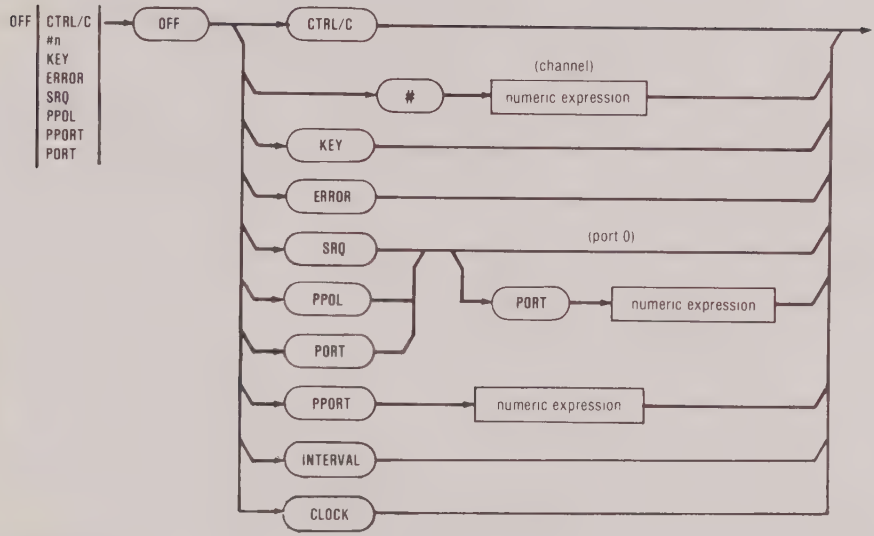
The OFF CTRL/C statement disables the action of a previous ON CTRL/C GOTO statement.

- An OFF CTRL/C statement in a CTRL/C interrupt processing routine will cause the Instrument Controller to return to Immediate Mode.
- After an OFF CTRL/C statement, the ABORT switch or a ⟨CTRL⟩/C keyboard entry will return the controller to Immediate Mode.

Usage

OFF #(expression)

Syntax Diagram



Description

The OFF #n statement disables the action of all previous ON #n GOTO statements for the specified channel.

- An OFF #n statement in a serial port interrupt processing routine will disable further interrupts on the referenced channel, but will not affect the interrupt processing that is in progress.
- The OFF #n statement does not close the referenced channel.
- If the channel is closed by a CLOSE n statement, further interrupts are disabled and the OFF #n statement is unnecessary.

Example

```
12345 OFF #2                    ! Disable RS-232 interrupts from
12346                           ! Channel 2
```

Statement

Usage

OFF CLOCK

Syntax Diagram

Description

The OFF CLOCK statement cancels any previously issued ON CLOCK interrupt.

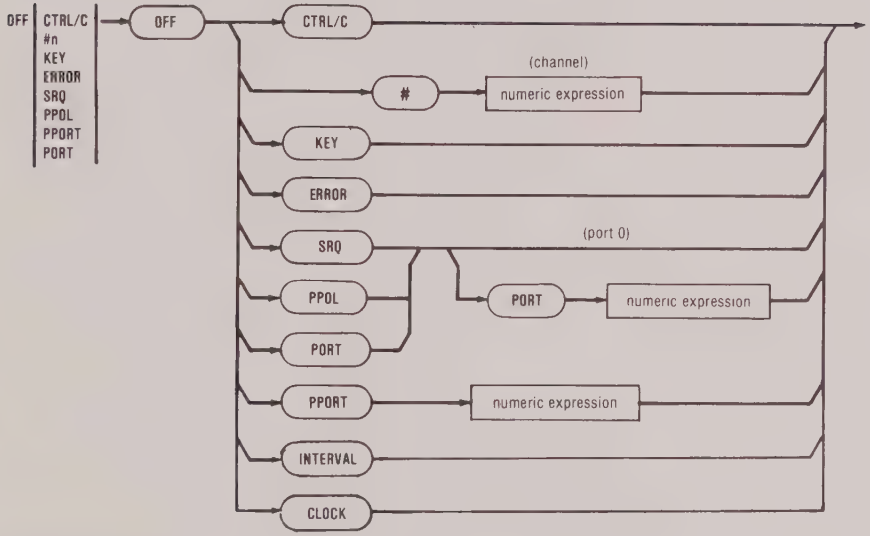
Remarks

Refer to the ON CLOCK statement.

Usage

OFF ERROR

Syntax Diagram



Description

The OFF ERROR statement disables the action of a previous ON ERROR GOTO statement.

- An OFF ERROR statement in an error processing routine will terminate the program.
- After an OFF ERROR statement has been executed, a level R (recoverable) error will terminate the program.
- After an OFF ERROR statement, a level W (warning) error will be ignored.

Example

```
43210 OFF ERROR                ! Suspend error interrupts
```

OFF INTERVAL 80
Statement

Usage

OFF INTERVAL

Syntax Diagram

Description

The OFF INTERVAL statement cancels any previously issued ON INTERVAL interrupt.

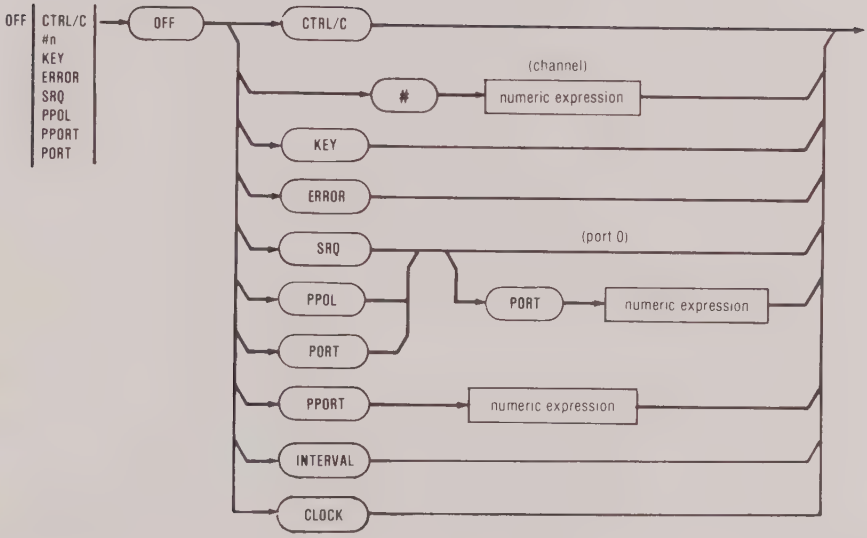
Remarks

Refer to the ON INTERVAL statement.

Usage

OFF KEY

Syntax Diagram



Description

The OFF KEY statement disables the action of a previous ON KEY GOTO statement.

- An OFF KEY statement in a key-interrupt processing routine will prevent the routine from being continuously reentered if the KEY buffer is not reset in the routine.
- OFF KEY disables further checking of the value of KEY.
- An OFF KEY statement in any interrupt-processing routine will not have any additional effect.

Example

```
13579 OFF KEY          ! Stop checking for key-entry interrupts
```

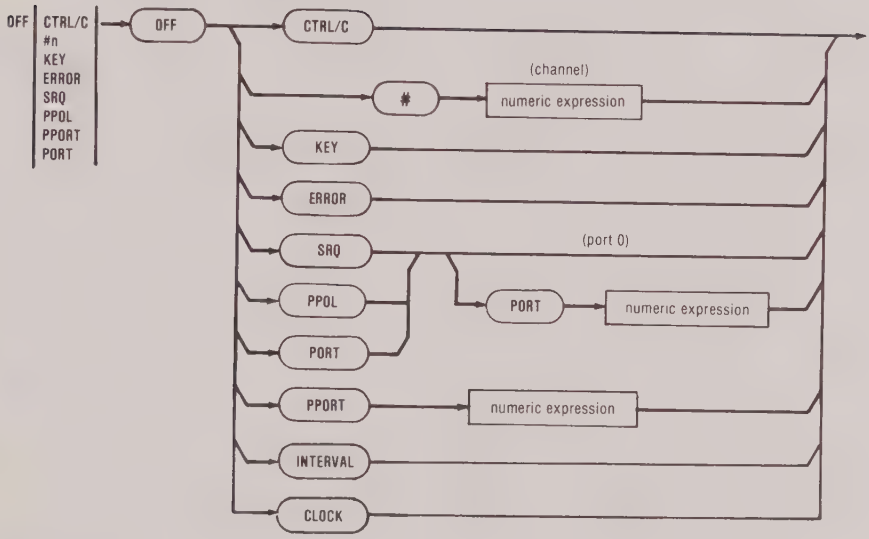
OFF PPOL Statement

IEEE-488

Usage

OFF PPOL

Syntax Diagram



Description

The OFF PPOL statement disables the action of a previous ON PPOL GOTO statement.

- An OFF PPOL statement in a parallel poll response routine will prevent the routine from being immediately reentered after the RESUME statement if the routine does not clear the instrument poll response bit.

- An OFF PPOL statement in any interrupt processing routine will not have any additional effect.

Example

```
12120 OFF PPOL                  ! Disable parallel poll response
```

OFF PPORT Statement

Usage

OFF PPORT {numeric expression}

Syntax Diagram

Description

The OFF PPORT statement cancels the action of a previous ON PPORT GOTO statement on the parallel port specified by the numeric expression.

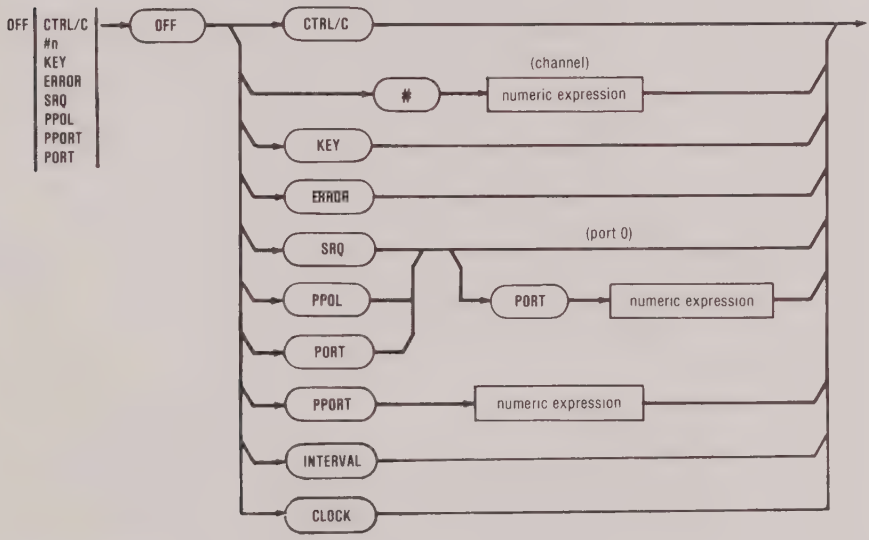
Remarks

Refer to the ON PPORT statement.

Usage

OFF SRQ [port {numeric expression}]

Syntax Diagram



Description

The OFF SRQ statement disables the action of a previous ON SRQ GOTO statement.

- An OFF SRQ statement in a service request processing routine will prevent the routine from being continuously reentered if it does not reset the service request by performing a serial poll.
- An OFF SRQ statement in any interrupt processing routine will not have any additional effect.

Example

```
9090 OFF SRQ   ! Disable further responses to Service Request
```

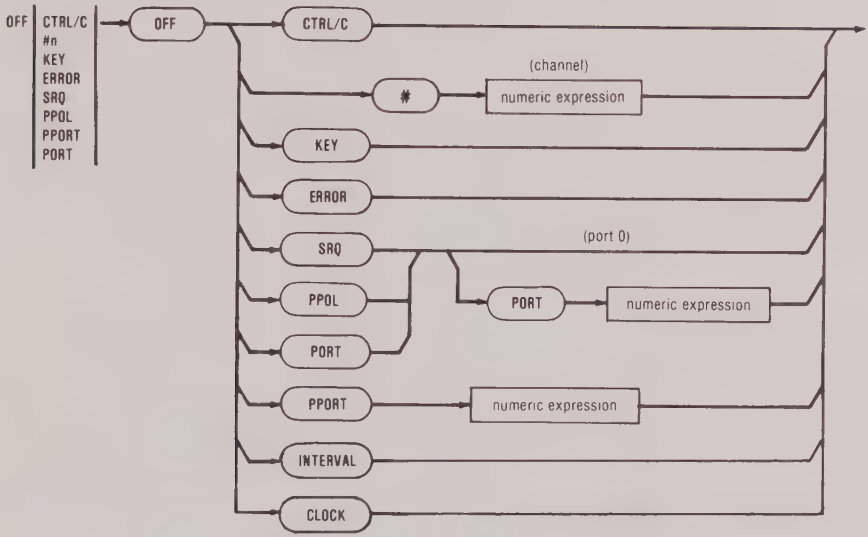
Statement

IEEE-488

Usage

OFF PORT[expression]

Syntax Diagram



Description

The OFF PORT statement disables the action of a previous ON PORT GOTO statement.

- Port 0 interrupts are disabled if no port expression is specified.

Example

```
9090 OFF PORT  ! Disable further responses to Port Status changes
```

Immediate Mode Command

Usage

OLD {pathname}

Syntax Diagram

Description

The OLD immediate mode command is used to load a program into memory from an input, device or file.

The pathname, including the optional storage device prefix, filename, and name extension, must be inclosed in quotes.

- OLD may only be used in Immediate mode.
- BASIC will look for the file on the System Device SY0: if a device name is not specified.
- BASIC will look for the file on a specified device if the device name is included as a file name prefix.
- This command assumes that the file named is a valid BASIC program in either ASCII or lexical form.
- If the file name extension is .BAS or .BAL, it does not need to be specified in the file name.
- If no extension is specified, BASIC looks for a file with .BAL name extension and loads that file if it exists.
- If the file named does not exist with a .BAL extension, BASIC looks for the file with a .BAS extension and loads it if it exists.
- If the file exists in both lexical (.BAL) and ASCII (.BAS) form, BASIC will load the lexical form unless the command directly specifies otherwise.
- The OLD command will save the name of a file to be used if a subsequent SAVE, SAVEL, RESAVE, or RESAVEL command is given for which the filename is omitted.

OLD
Immediate Mode Command

Example

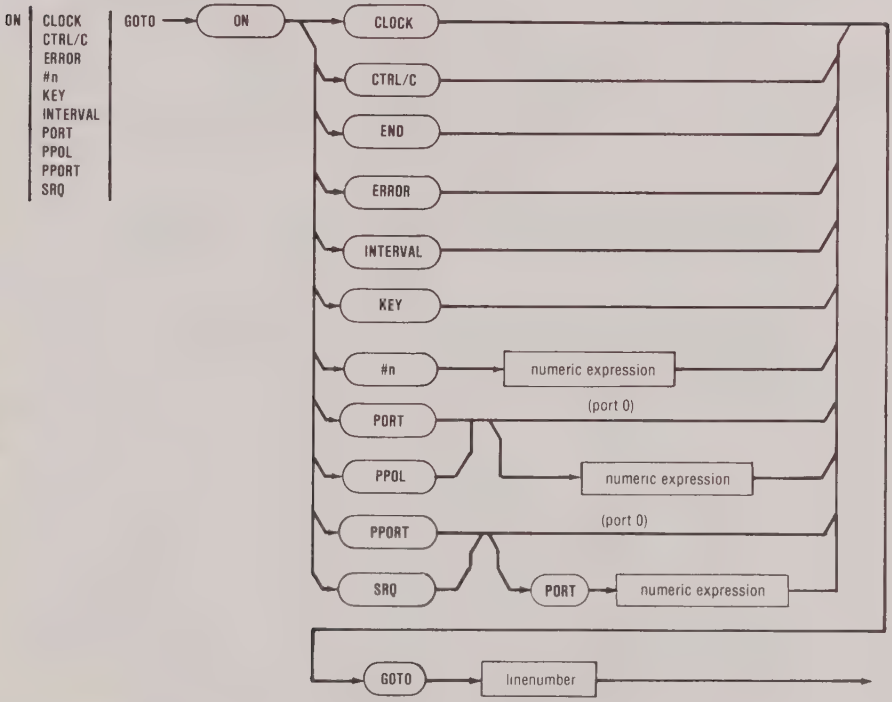
The following examples illustrate use of the OLD command:

COMMAND	RESULT
OLD "TEST"	Load the file named TEST.BAL (if present) or TEST.BAS (if TEST.BAL is not present) from the default System Device into memory.
OLD "MF0:TEST.5"	Load the file named TEST.5 from the floppy disk.
OLD "TRAC.TOR"	Load the file TRAC.TOR from the System Device.
SAVEL	Save a lexical version of TRAC.TOR in TRAC.BAL.

Usage

ON CLOCK GOTO {linenumber}

Syntax Diagram



Description

The ON CLOCK GOTO statement is used to create an interrupt at a specific time of day.

- The time must be set using the SET CLOCK statement prior to using the ON CLOCK GOTO statement.
- The ON CLOCK interrupt remains in effect until an OFF CLOCK statement is executed.
- A RESUME statement is required to terminate the clock interrupt handler.

ON CLOCK
Statement

Example

This simple example

```
SET CLOCK 18:00                 ! Set the clock to 6:00 PM
ON CLOCK GOTO 12000             ! And enable an interrupt at that time
```

will set an interrupt at six o'clock every evening.

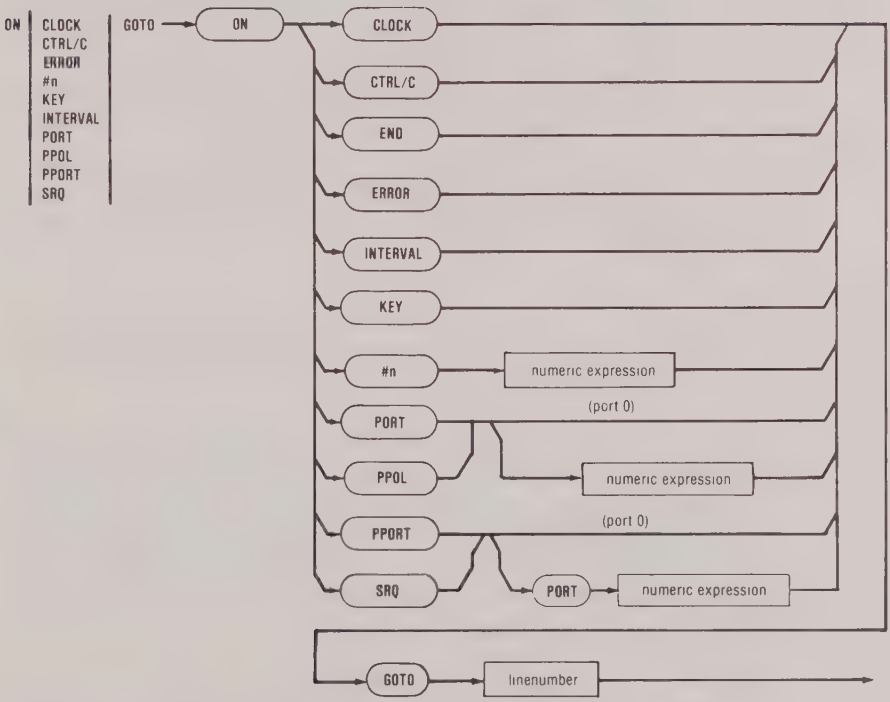
The following example can be used to reschedule a CLOCK interrupt every hour:

```
11010 NH = INT(TIME / 24.0) + 1.0          ! Define Next_Hour
11020 IF NH > 23.0 THEN NH = 0.0
11030 SET CLOCK NH                         ! Set clock time
11040 ON CLOCK GOTO 12010
.
.
.
12010 ! Clock interrupt handler
12020 NH = INT(TIME / 24.0) + 1.0          ! Redefine Next_Hour
12030 IF NH > 23.0 THEN NH = 0.0
12040 SET CLOCK NH                         ! Reschedule interrupt
12050 ! Other statements
12060 RESUME
```

Usage

ON CTRL/C GOTO linenumber

Syntax Diagram



Description

The ON CTRL/C GOTO statement enables a program to respond to a random occurrence of an ABORT switch or a ⟨CTRL⟩/C keyboard entry by transferring control to a specified program routine containing a user-defined response.

- When an ABORT switch or a CTRL/C keyboard entry is detected, control transfers to the specified line number.
- The CTRL/C handling routine must explicitly acknowledge the interrupt with a RESUME statement.

ON CTRL/C
Statement

- BASIC normally responds to the ABORT switch or a ⟨CTRL⟩/C keyboard input by terminating the program and returning to Immediate Mode. This statement alters the normal interpreter response.

- When a CTRL/C has been detected, further checking for interrupt conditions other than ERROR is suspended until RESUME is encountered.

- If a second CTRL/C is detected before encountering a RESUME statement, it is ignored.

- A RESUME statement will return control to execute the first statement not completed when the CTRL/C or ABORT key entry was detected.

NOTE

Further interrupt processing is suspended until a RESUME statement is performed.

If a level R or W (recoverable or warning) error occurs after CTRL/C detection and before encountering a RESUME statement, CTRL/C processing is suspended either temporarily or permanently. If the program includes error processing, control will be returned to the CTRL/C processing routine when error processing is completed. Without error processing, a level W error is ignored.

NOTE

Since CTRL/C is the only way to manually stop a BASIC program without deleting it, if not handled properly, a CTRL/C interrupt to an ON CTRL/C subroutine can lock a program into Run Mode.

When the keyboard is in the "noecho" mode (see the SET NOECHO statement), a CTRL P entry will also transfer control to the CTRL/C handler.

Example

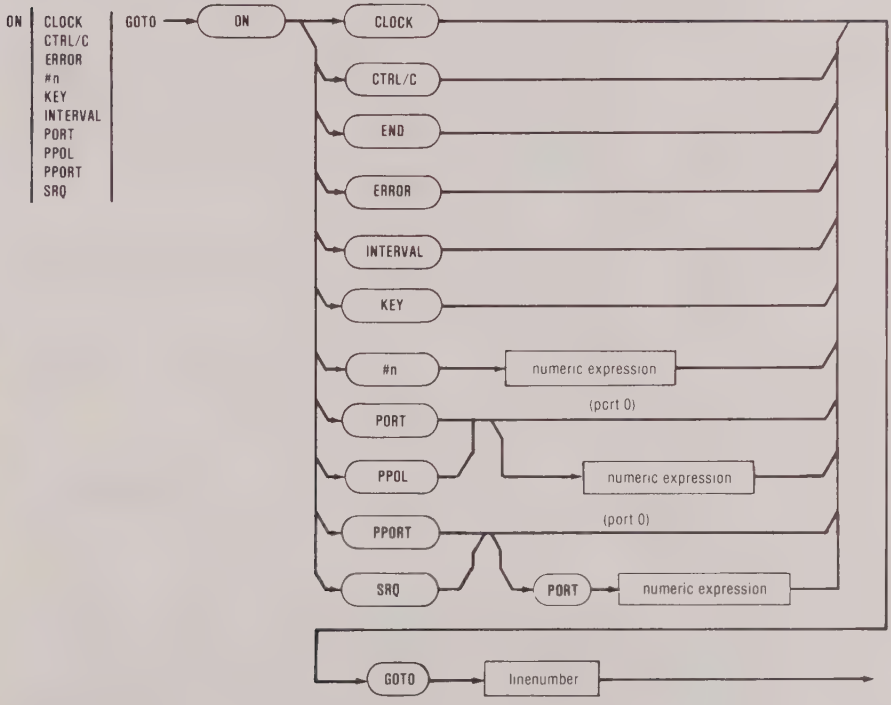
The following example shows portions of a program used to create a CTRL/C interrupt handler.

```
140 ON CTRL/C GOTO 13800        ! Set up vector for ^C int. handler
150 ! Other statements
.
.
.
13790 ! ^C Interrupt handler
13800 ! ^C handling statements
.
.
.
15000 RESUME                    ! Continue with the program
```

Usage

ON ERROR GOTO {linenumber}

Syntax Diagram



Description

The ON ERROR GOTO statement enables a program to respond to a random occurrence of an error condition by transferring control to a specified routine containing a user-defined response.

- When an error is detected, control transfers to the specified line number immediately.
- The program section following the specified line number must explicitly acknowledge the interrupt with a RESUME statement.

- Only level R and W (recoverable or warning) errors can be processed by an error routine. Level F (fatal) errors always terminate the program.

- Level R interrupts will terminate a program unless an ON ERROR GOTO statement has been executed so that the error condition can be treated.

- Without error processing, a level W error is ignored.

- When an error condition has been detected, further checking for interrupt conditions other than ERROR or ⟨CTRL⟩/C is suspended until a RESUME is executed.

- When an error condition has been detected, the system variable ERL will contain the line number at which the error occurred, and ERR will contain the error number.

- If a second error is detected before encountering a RESUME statement, the program terminates immediately.

NOTE

Further interrupt processing is suspended until a RESUME statement is performed.

- If a ⟨CTRL⟩/C interrupt occurs after error detection and before encountering a RESUME statement, error processing is suspended either temporarily or permanently. If the program includes ⟨CTRL⟩ C interrupt processing with a RESUME statement, control will be returned to the error-processing routine when ⟨CTRL⟩/C processing is completed.

- A RESUME statement that does not specify a linenumber will return control to re-execute the statement that caused the error.

Example

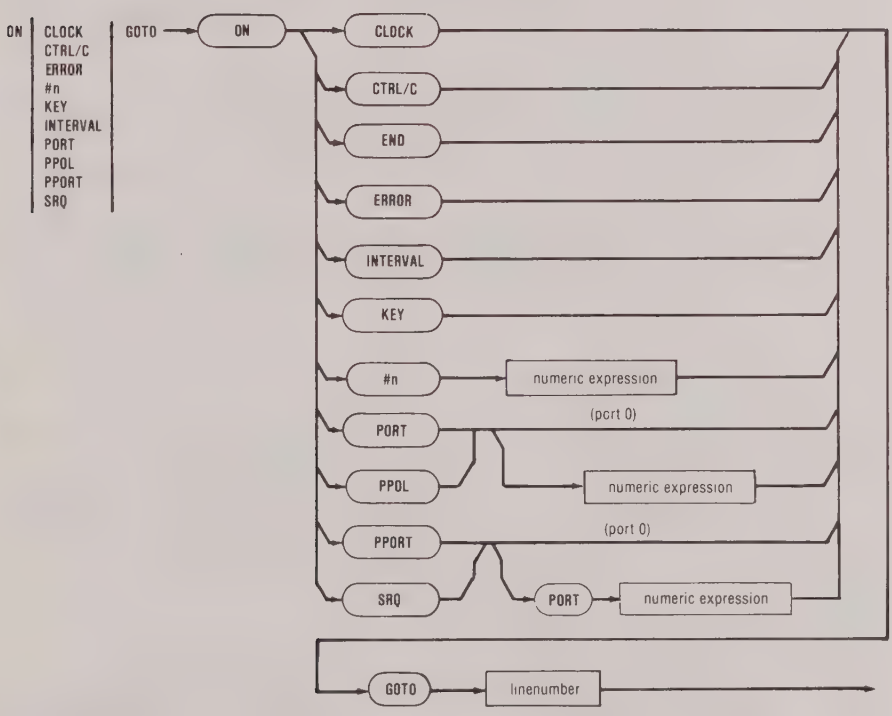
The code segments below illustrate how to create a user-defined Error Handler.

```
120 ON ERROR GOTO 8990          ! Create an error-handling vector
130 ! Other statements
.
.
.
8990 ! Error-handling routine
9000 IF ERL( )12045 AND ERR( )300 THEN 9020 ! Look for a Disk Not
9001                                        ! Loaded error at line 12045
9010 PRINT "Insert a disk and try again"    ! and display some help.
9020 RESUME
.
.
.
```

Usage

ON #{expression} GOTO {linenumber}

Syntax Diagram



Description

The ON #n GOTO statement enables a program to respond to End-Of-Line or End-Of-File characters received through an open input channel from a serial (RS-232-C) port. Control is transferred to the specified program routine containing a user-defined response whenever either terminator character is received.

- When a terminator character is received through an open input channel from a serial (RS-232-C) port, control is transferred to the specified line number after completion of the current statement.

- End-Of-Line and End-Of-File terminator characters are defined by the SET utility program.

- The program section following the specified line number must explicitly acknowledge the interrupt with a RESUME statement. Program control then resumes at the next statement following the one that was completed when the terminator character was received.

- When a terminator character has been detected, further checking for interrupt conditions other than ERROR or ⟨CTRL⟩ C is suspended until the RESUME statement is encountered.

- If this statement is used more than once in a program with the same channel number, control is transferred to the line number referenced in the most recently encountered ON #n GOTO statement.

- Any of the sixteen available user channels may be used.

- The referenced channel must be opened for input (OLD or not specified) prior to the ON #n GOTO statement.

- When interrupts occur on more than one channel simultaneously, the lowest numbered channel has highest priority.

Example

```
100 ! Set up two RS-232 channel interrupt vectors
110 ON #2 GOTO 21000                ! Channel 2 handler @ 21000
120 ON #3 GOTO 26000                ! Channel 3 handler @ 26000
130 ! Other statements

21000 ! Channel 2 interrupt handler
21010 ! Other statements
21020 RESUME

26000 ! Channel 3 interrupt handler
26010 ! Other statements
26020 RESUME
```

ON...GOTO
ON...GOSUB
Statement

Usage

ON {expression} GOTO {line list}
ON {expression} GOSUB {line list}

Syntax Diagram

Description

The ON GOTO and ON GOSUB statements define multiple control branches. Control transfers to on of the lines listed in the statement, depending upon the current value of the expression.

- The expression must be numeric.
- The expression is evaluated and rounded to obtain an integer.
- The integer is used as an index to select a line number from the list contained in the statement.
- The range of the integer must be between 1 and the number of line numbers contained in the list.
- The branch of control may be either a GOTO transfer (refer to the GOTO statement) or a subroutine call (see the GOSUB statement).

Example

The following example illustrates a common use of ON-GOTO:

Statement	Meaning
ON A GOTO 400, 500, 600	Transfers control to line 400 if A = 1, to line 500 if A = 2, or to line 600 if A = 3.

The following example illustrates one use of ON-GOTO at line 1030. Since the expression is normally rounded, INT(LOG(R)) is used instead of LOG(R) to ensure that the voltage divider is used for all values less than 1 volt (1000 millivolts).

ON...GOTO
ON...GOSUB
Statement

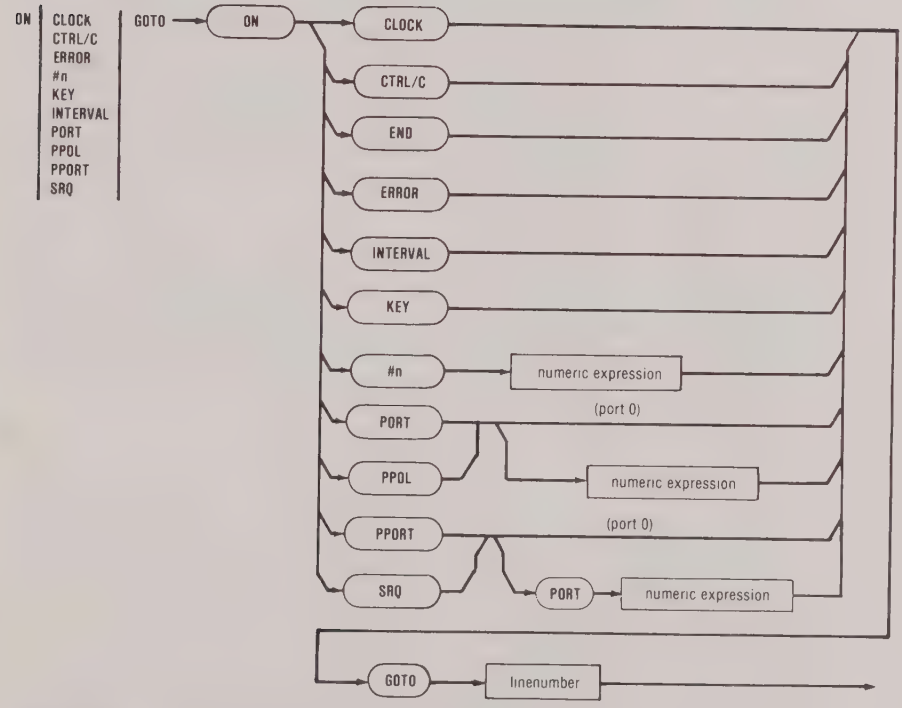
```
1000 REM -- Connect Instrument to Test Station
1010 ! R on input is the value of the voltage to be applied
1020 ! R is in millivolts and is between 10 and 10^6
1030 ON INT(LOG (R)) GOTO 1200,1200,1300,1300,1400,1400
1200 REM -- Setup External Voltage Divider
1210 ! Other statements
1290 GOTO 1500
1300 REM -- Connect Instrument Directly
1310 ! Other statements
1390 GOTO 1500
1400 REM -- Give High Voltage Warning and Then Connect Instrument
1410 ! Other statements
1490 GOTO 1500
1500 REM -- Take Readings
1510 ! Other statements
```

ON INTERVAL Statement

Usage

ON INTERVAL GOTO {line number}

Syntax Diagram



Description

The ON INTERVAL statement activates an interrupt at previously specified periodic interval. An interval interrupts is enabled by:

1. Using a SET INTERVAL statement to specify the interval period to be used, and

2. Arming the interrupt and selecting an interrupt handler by executing an ON INTERVAL statement.

The intervals are calculated so that the execution time of the interrupt handler will not affect the timing. Assume that the statement:

```
100 SET INTERVAL 0:0:1 ! one second
```

has been executed. If the interrupt handler takes 20 milliseconds to execute, the next interval interrupt will occur one second after the last interrupt, not one second plus 20 milliseconds after the last interrupt.

The intervals are calculated so that the execution time of the interrupt handler will not affect the timing. If the interrupt handler takes 20 milliseconds to execute, the next interval interrupt will occur one second after the last interrupt, not one second plus 20 milliseconds after the last interrupt.

If the interval time is short and a statement (say, an INPUT statement from an instrument) takes longer than the timing interval, the interval interrupt will occur as soon as the long statement is finished. Subsequent interval interrupts will continue to occur at the same timing intervals as before. Refer to the timing diagram:

Interval interrupts occur at the vertical arrows.

Interval interrupts occur at the x's.

The ON INTERVAL statement may be disabled by executing an OFF INTERVAL statement.

Example

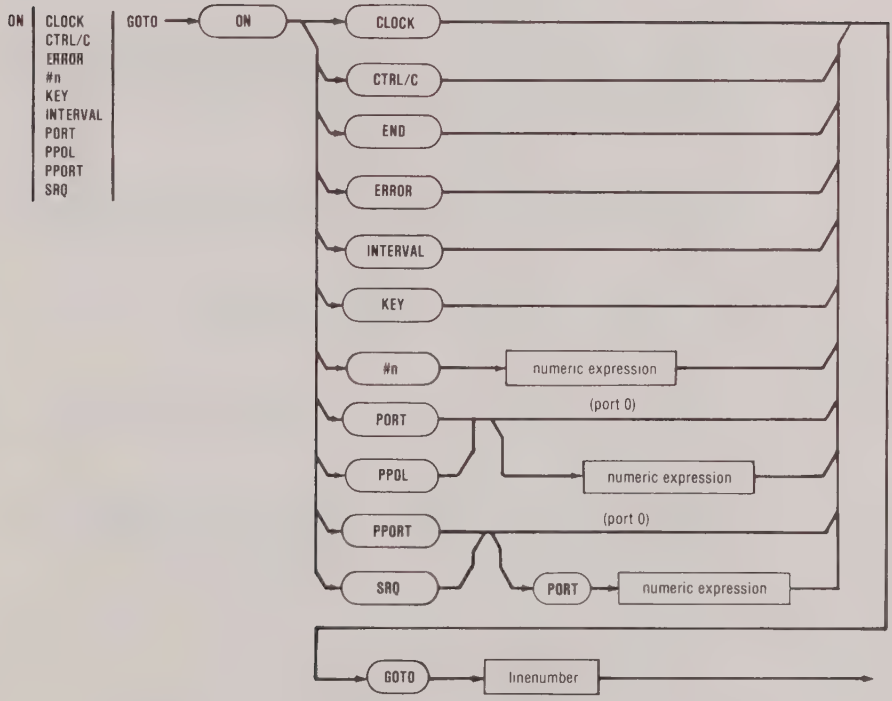
A periodic (interval) interrupt is activated by:

```
120 SET INTERVAL 1500              ! set interrupt every 1500 ms
130 ON INTERVAL GOTO ...           ! activate interrupt
```

Usage

ON KEY GOTO {line number}

Syntax Diagram



Description

The ON KEY statement enables a program to respond to the occurrence of a key entry on the touch-sensitive display by transferring control to a specified program routine containing a user-defined response.

- When a key entry is detected, control transfers to the specified line number after completion of the current statement.
- The program section following the specified line number must explicitly acknowledge the interrupt with a RESUME statement.

ON KEY
Statement

- When a KEY entry has been detected, further checking for interrupt conditions other than ERROR or CTRL/C is suspended until RESUME is encountered.

- When a KEY entry has been detected, the system variable KEY will contain the number of the last touch key pressed.

- The system variable KEY is set whenever the Touch-Sensitive Display is pressed in an active area, regardless of whether ON KEY GOTO is used. It remains set until it is read by a program statement. (For example, K% = KEY)

- If the system variable KEY is non-zero when ON KEY GOTO is executed, control is immediately transferred to the specified line number.

- ON KEY does not reset the system KEY variable.

Example

```
120 ON KEY GOTO 1200                ! Activate key interrupt
.
.
.
1200 ! Key interrupt handler
1210 K% = KEY
1215 IF K% = 20 GOTO 1300           ! Look for key number 20
1220 IF K% = 40 GOTO 1350           ! Look for key number 40
1230 PRINT "Try again"              ! Print notice
1240 RESUME                         ! And try again
```

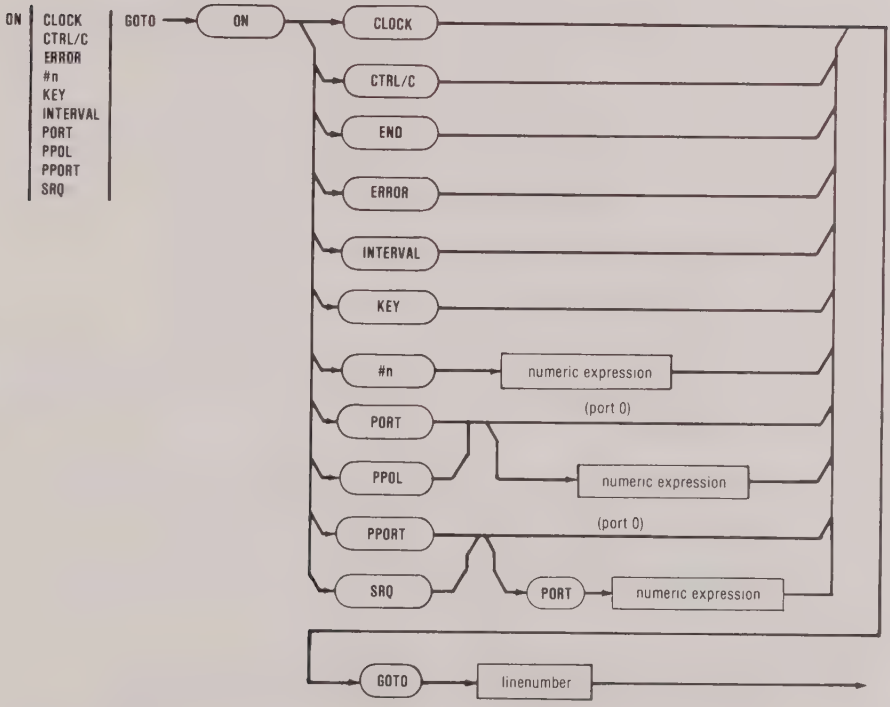
ON PORT Statement

IEEE-488

Usage

ON PORT [PORT p%] GOTO {linenumber}

Syntax Diagram



Description

The ON PORT statement enables a program to respond to an IEEE-488 port status change by transferring control to a specified program routine containing a user-defined response.

- When a status change is detected, control transfers to the specified line number after completion of the current statement.
- The program section following the specified line number must explicitly acknowledge the interrupt with a RESUME statement.

ON PORT
Statement

IEEE-488

- When a status change has been detected, further checking for interrupt conditions other than ERROR or CTRL/C is suspended until RESUME is encountered.
- One interrupt vector is permitted for every configured port.
- When the port expression is omitted, port 0 is used.
- The port number given in the port expression must be in the range of [0..1].
- Floating-point port numbers will be rounded to an integer.
- A status change interrupt is triggered whenever:
  1. A DCL command is received by the controller,
  2. A GET command is received by the controller,
  3. The first serial poll of the controller (after a SET SRQ statement has been executed) is performed by the Controller in Charge,
  4. A change in the state (i.e., idle, controller, talker, listener) of the controller IEEE-488 interface driver occurs.

The PORTSTATUS() function may used to determine the exact cause of the interrupt.

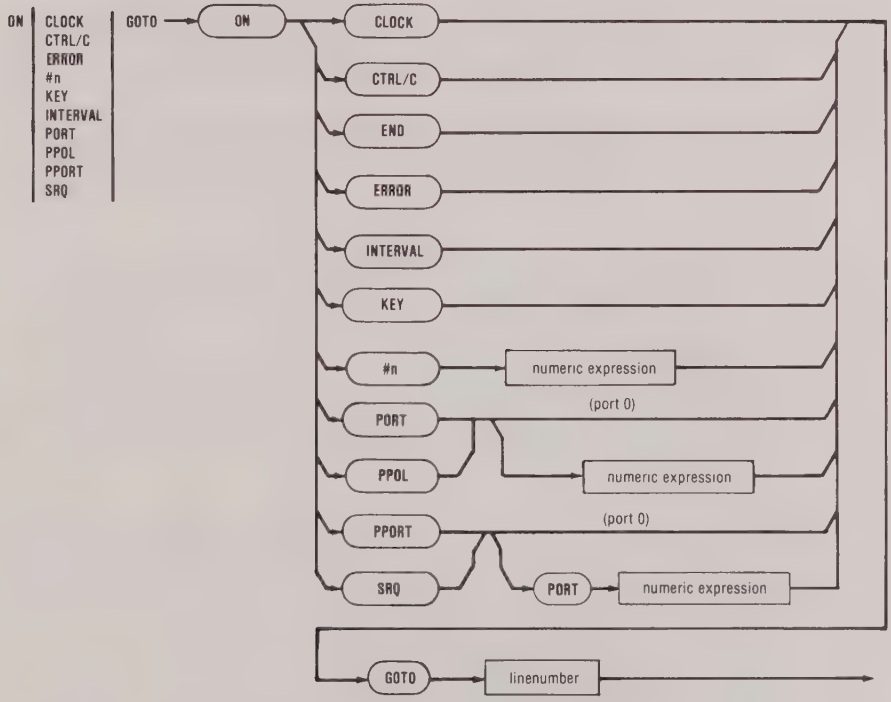
Example

```
130 ON PORT GOTO 1000          ! Activate port 0 interrupt
140 ON PORT 1 GOTO 1900        ! Activate port 1 interrupt
.
.
1000 ! Port 0 interrupt handler
1000 ST% = PORTSTATUS(TP%)                  ! read port's status
1010 SA% = ST% AND 3%                       ! isolate controller state
1020 TR% = ST% AND 16%                      ! triggered?
1030 CL% = ST% AND 8%                       ! received DCL or SDC?
1040 PL% = ST% AND 4%                       ! was SRQ acknowledged?
1050 IF TR% THEN ...                        ! action for GET command
1060 IF CL% THEN ...                        ! action for clear command
1070 IF PL% THEN ...                        ! action for SRQ acknowledge
1080 ON SA% + 1% GOTO 1090, 110, 1110, 1120
1090                                        ! code for idle port
...
1100                                        ! code for Controller
...
1110                                        ! code for Listener
...
1120                                        ! code for Talker
...
.
1900 ! Port 1 interrupt handler
```

Usage

ON PPOL [PORT p%] GOTO {line number}

Syntax Diagram



The ON PPOL GOTO statement enables a program to respond to a positive parallel poll response from a configured instrument by transferring control to a specified program routine containing a user defined response.

- The ON PPOL GOTO statement initiates parallel polling on the specified port, or on Port 0 if not specified. A poll will be performed following the completion of each BASIC statement.

ON PPOL
Statement

IEEE-488

- When positive response to a parallel poll is detected, control transfers to the specified line number after completion of the current statement.

- The program section following the specified line number must explicitly acknowledge the interrupt with a RESUME statement.

- When a positive response to a parallel poll has been detected, further checking for interrupt conditions other than ERROR or CTRL/C is suspended until RESUME is encountered.

- If both Port 0 and Port 1 have PPOL interrupts enabled, port 0 will be checked for a parallel poll response prior to checking Port 1.

NOTE

Some instruments clear a parallel poll bit when the condition causing it disappears, or when the bus port is parallel polled.

When possible, the instrument responding to the parallel poll should be programmed within the processing routine to reset its poll response bit. If this bit remains set, the routine will be immediately reentered after the RESUME statement.

Example

In the following example, line 10 indicates that the PPOL handling routine starts at line 1000. At line 550, the PPOL is disabled by OFF PPOL, and the program is halted by STOP.

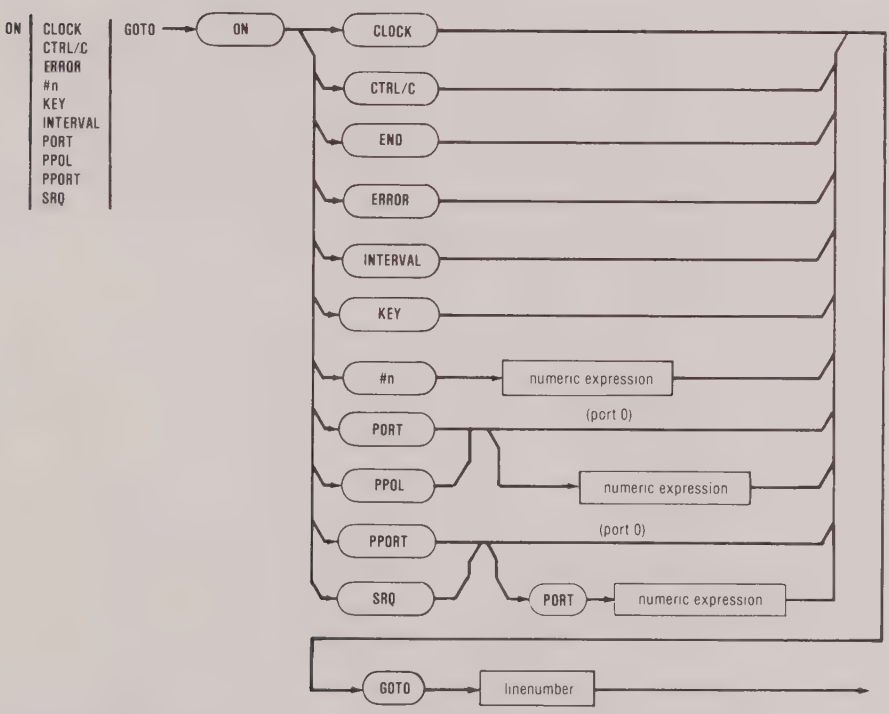
```
10      ON PPOL GOTO 1000              ! Poll port 0
20      ! other statements
30      !
550     OFF PPOL \ STOP
1000    ! Port 0 PPOL handler
1010    ! other statements
1020    !
1070    RESUME                         ! Return from PPOL handler
```

ON PPORT Statement

Usage

ON PPORT [PORT p%] GOTO {line number}

Syntax Diagram



Description

The ON PPORT statement enables a program to respond to data from the optional Parallel Port by transferring control to a specified program routine containing a user-defined response.

- When data is detected, control transfers to the specified line number after completion of the current statement.
- The program section following the specified line number must explicitly acknowledge the interrupt with a RESUME statement.

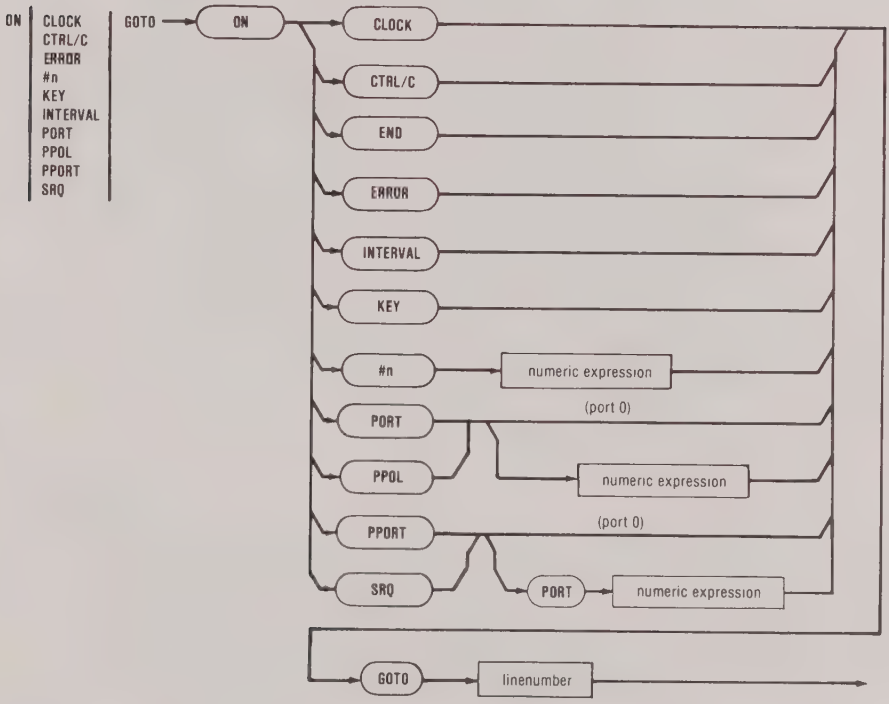
ON PPORT
Statement

- When data has been detected, further checking for interrupt conditions other than ERROR or CTRL/C is suspended until RESUME is encountered.
- One interrupt vector is permitted for every port.
- When the port expression is omitted, port 0 is used.
- The port number given in the port expression must be in the range of [0..15].
- Floating-point port numbers will be rounded to an integer.
- An interrupt is triggered whenever a data bit/byte/word is received on the specified Parallel Port.

Usage

ON SRQ [PORT P%] GOTO {line number}

Syntax Diagram



Description

The ON SRQ GOTO statement enables a program to respond to the occurrence of a service request from an instrument by transferring control to a specified program routine containing a user defined response.

- The specified port is sampled after the completion of each statement. If a port is not specified, port 0 is sampled.
- When a service request is detected, control transfers to the specified line number after completion of the current statement.

ON SRQ
Statement

- The program section following the specified line number must explicitly acknowledge the interrupt with a RESUME statement.

- When a service request has been detected, further checking for interrupt conditions other than ERROR or CTRL/C is suspended until RESUME is encountered.

- An internal SRQ flag is set by a service request on the enabled port. It is reset in the controller by performing a serial poll on any instrument on the port requesting service (for example, Y% = SPL(10)). However, depending on the instrument, SRQ will probably be set again until the instrument requesting service is serial polled. This will cause the service request routine to be immediately reentered after the RESUME statement.

- ON SRQ GOTO does not reset the internal SRQ flag.

- When SRQs are present on both port 0 and port 1 simultaneously, the SRQ on port 0 will be responded to first.

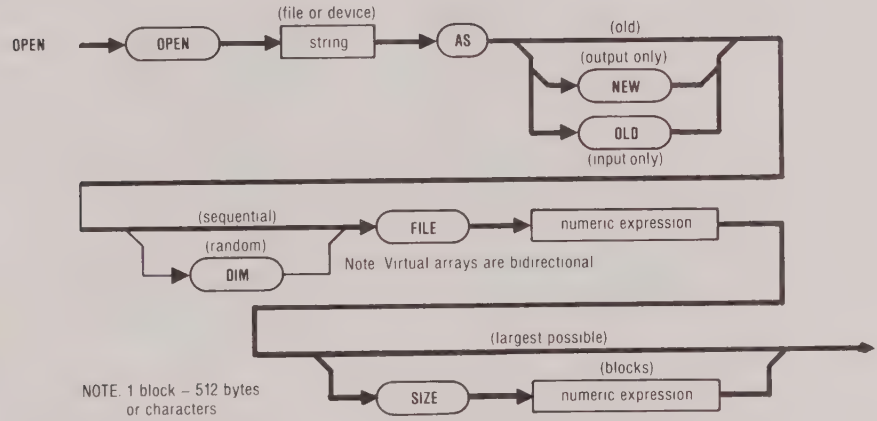
- Separate SRQ vectors may be set for each port.

Statement

Usage

OPEN {filename$} AS [NEW or OLD] FILE {channel%} [SIZE blocks]

Syntax Diagram



Description

The OPEN statement assigns a data communication channel numbered between 1 and 16 to a file or device. The following points apply to any use of the OPEN statement:

- All subsequent input from and output to the file or device is made by reference to the channel number.
- The channel is sequential access, unless it is a virtual-array channel. Data is sent to, or retrieved from, sequential channels in serial order.
- A virtual-array channel, requiring the DIM specification, is random access. Data is sent to, or retrieved from, virtual arrays in any order.
- The string following OPEN indicates the name of the file or device. The default extension for file names is ". " (3 spaces).

The AS NEW, AS OLD, and DIM optional constructions of the OPEN statement indicate specific and different actions for sequential and random channels. The following points discuss these differences:

- AS must be specified. If it is not followed by NEW or OLD, OLD is assumed.
- DIM specifies a random-access virtual-array file.
- For a sequential channel, NEW indicates an output channel. OLD, or no specification, indicates an input channel.
- For a random-access virtual-array channel, NEW indicates that an existing file by the specified name is to be deleted, if there is one.
- For a random-access virtual-array channel, OLD (or no specification) indicates that an existing file by the specified name is to be accessed. If the file is not found, error 305 results.
- Random-access virtual-array channels are bidirectional.

The numeric expression following FILE indicates the channel number to be assigned. Following are some points to be considered in selecting a channel number:

- The value of the numeric expression must be between 1 and 16.
- Each channel number can only be used for one operation at a time.
- A channel that was previously opened for a different purpose, and is no longer in use, must first be closed before being reassigned.
- SIZE has significance only for NEW files. It is ignored for OLD files. The following points discuss some of the implications of the SIZE specification:
- The numeric expression following SIZE specifies the number of 512 character blocks to be reserved for a new file.
- A new file opened without a size specification will be assigned the largest available contiguous area.

- The first new file opened on a device is assigned all available free space if there is only one free area available. An attempt to open another new file on the same device, before the first new file is closed, results in error 306 (no room on device).

- SIZE may be used to limit the amount of space reserved so that a second new file can be opened on the same device.

- SIZE may also be used to verify that a new file will have required space available. For example, if a file requires 40 blocks, the statement:

- OPEN "ED0:TEST.ARC" AS NEW FILE 1% SIZE 40%

- Will cause error 306 if 40 contiguous blocks are not available on the Electronic Disk (device name ED0:).

- The SIZE of a sequential file is the number of 512 character blocks needed to hold the file.

Examples

```
10 OPEN "TEST.ARY" AS NEW FILE 1 SIZE 30
20 OPEN "TEST.OLD" AS FILE 2
30 OPEN "KB0: AS NEW FILE 15
```

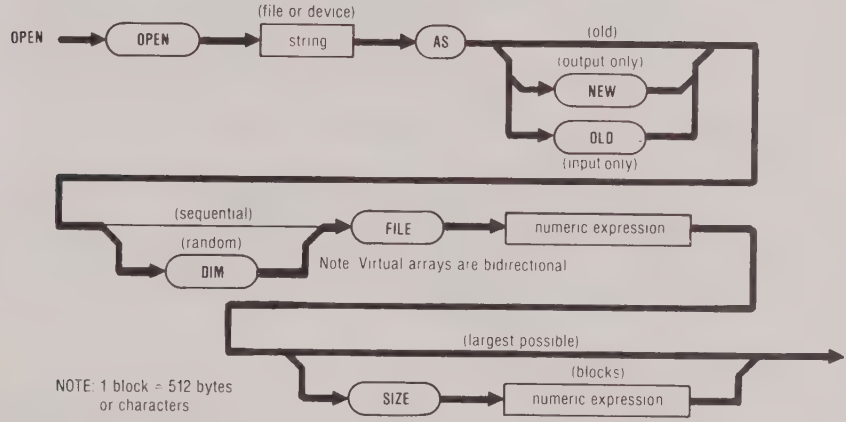
Remarks

The use of the OPEN statement for virtual arrays is discussed elsewhere in this section.

Usage

OPEN {filename$} AS [NEW or OLD] DIM FILE {channel%} [SIZE blocks]

Syntax Diagram



Description

OPEN assigns a channel number to one or more virtual arrays (random access files).

- A file name specification must follow OPEN.
- The file will be on the default System Device (SY0:) unless a device specification is included with the file name.
- A virtual array may only exist on a file-structured device due to the random access requirement.
- Unlike ordinary channels described, virtual-array channels are bidirectional.
- NEW specifies that the file will replace any existing file found with the specified name.
- OLD specifies that the channel will be associated with an existing file. Error 305 results if the file cannot be found.

- OLD is assumed when neither NEW nor OLD is specified.
- DIM must be included to specify that the channel opened will be a random-access file.
- The numeric expression following FILE designates the channel number.
- Channels are integer numbers 1 through 16.
- Error 303 results if the channel is already in use.
- The numeric expression following SIZE specifies the number of 512-byte blocks to be allocated for a NEW file.
- The largest available contiguous space is allocated for a NEW file when SIZE is not specified.
- SIZE is ignored for OLD files.
- SIZE may need to be specified to open two or more NEW files simultaneously on the same device. If the device is packed, or has not had any file deletions, there is only one contiguous space available.
- Each floating-point element occupies 8 bytes (64 bits).
- Each integer element occupies 2 bytes (16 bits)
- Each string element occupies 16 characters, unless specified otherwise by the DIM statement.
- When the channel is closed, information supplied by the DIM statement is used to reduce the space allocated to the virtual array to that actually required.

Example

The following program fragment illustrates the use of the OPEN statement, as applied to virtual arrays (random access files).

```
10   OPEN "ARRAY.VRT" AS OLD DIM FILE 1 SIZE 10
20   OPEN "TEST.BIN" AS NEW DIM FILE 2
30   ! more code
40   ! more code
50   DIM #1, AR$(9,9)=32, BB%(959)
60   DIM #2, TE(199)
70   ! more code
          .
          .
          .
```

Remarks

Refer to the DIM statement for additional examples of declaring virtual arrays.

Statement

Usage:

PACK [device$]

Syntax Diagram:

Description

The PACK statement is used to reorganize a file-structured device to consolidate unused areas. When files are deleted, blank areas are left within the file structure. The PACK statement compacts these areas into one contiguous file space. It may be possible to make room for a file that wouldn't otherwise fit by packing the device.

- Note that this statement may take some time to be completed; CTRL/C and CTRL/P are disabled while this statement is in progress.

- The device specification string must be enclosed in matching quotes.

- If this statement is issued in immediate mode with no command file active the message

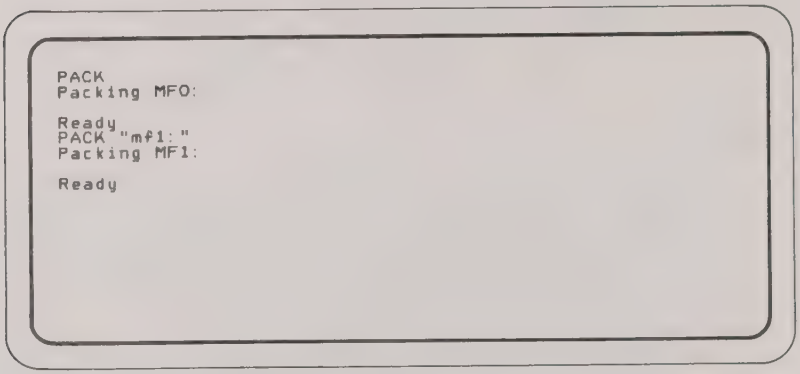
  Packing XXX:

  is printed to alert the user that a lengthy operation is in progress. The message will not appear if the PACK statement is executed by a program or during the execution of an immediate mode command in a command file. Packing "sy0:" will result in the actual device name being printed instead of "SY0:".

PACK
Statement

Example

The following display dialog illustrates using the PACK statement in the immediate mode:



In the first example, the PACK statement was used in the immediate mode to pack the System Device (since no device was specified). The System Device was MF0:.

In the second example, MF1: was specified and packed.

Statement

IEEE-488

Usage

PASSCONTROL device

Syntax Diagram

Once the CIC function has been passed to a different controller on the bus, the Controller may regain control of the bus in one of two ways:

- If the Controller is the "System Controller" (that is, the SYC switch is set true on the SBC) it can use the INIT statement. This sets the IFC (InterFace Clear) line and causes any other Controller-In-Charge (CIC) to relinquish control to the System Controller. Usually this form of "seizing control" is only done as a last resort.

- If the 1722A is not a System Controller and performs a Passcontrol, it can only regain control by having a Controller In Charge pass control back to it.

Description

The PASSCONTROL statement permits the Instrument Controller to designate another IEEE-488 bus instrument to act as Controller in Charge of the interface.

Example

The statement:

```
PASSCONTROL @23
```

addresses device 23 as a talker and issues a "take control" command to it. Runtime errors diagnosed are those which might occur with any addressed command (e.g., illegal device address, not CIC, etc.).

System Constant

Format

PI

Description

The system constant PI stores the value 3.14159265358979. PI represents the ratio of the circumference to the diameter of a circle.

- PI is stored as a floating-point constant
- PI may be used as any other floating-point variable, except that other values may not be assigned to it.

Example

The following example program computes the circumference of a circle.

```
10 ! compute the circumference of a circle
20 PRINT "diameter " \ INPUT D
30 C = PI * D
40 PRINT "The circumference is: ";C
50 END
```

PORT Expression 102A
Statement Modifier

Usage

PORT {numeric expression}

Syntax Diagram

Description

A PORT expression modifies a Fluke BASIC statement to communicate with an instrument or group of instruments after they have been previously addressed, without re-addressing them. Using a PORT expression as part of a BASIC statement suppresses individual device addresses and communicates directly with the named port.

The IEEE-488 port(s) are addressed by BASIC with the PORT statement modifier (port expression). A port expression takes the form:

```
PORT {numeric expression}
```

- The numeric expression used in a statement must evaluate to either 0 or 1. Refer to the 17XX Instrument Controller System Guide for identification of bus ports and information on cable connections.

- Once a device has been addressed, a PORT expression may be used to communicate directly with the device, bypassing any device addressing. For example, the statement:

```
PRINT @4, A$
```

  prints string A$ to device number 4 on IEEE-488 port 0. Now the same device may be addressed in a subsequent statement with:

```
PRINT PORT 0, A$
```

  This form of the PRINT statement sends the string A$ to port 0 without performing any device addressing.

Function

IEEE-488

Format:

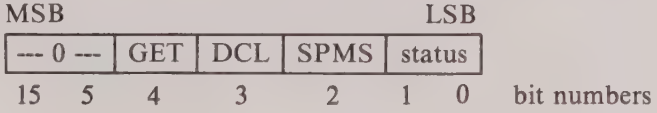
PORTSTATUS(p%)

Description

The PORTSTATUS() function determines the status of an interface port.

- PORTSTATUS() returns an integer containing the port status code for the specified port.
- The port number must be either 0 or 1. A floating-point value will be truncated to its integer value.

The PORTSTATUS() function returns an integer result having the following format:



The fields are:

GET is a single bit set nonzero if a GET (Group Execute Trigger) command has been received since the last time the PORTSTATUS() function was used.

DCL is a single bit set nonzero if a DCL (Device Clear) command has been received since the last time the PORTSTATUS() function was used.

SPMS is a single bit which indicates that a serial poll of the controller has been performed by the Controller in Charge since a SET SRQ statement was executed. This indicates that the SRQ generated by the BASIC program has been acknowledged by the Controller in Charge.

PORTSTATUS()
Function

IEEE-488

status is a two-bit integer which encodes the current state of the IEEE-488 interface port. The status should be interpreted as follows:

0 means that the interface is idle (the controller is not the Controller in Charge, not addressed as a listener, and not addressed as a talker). The status of an unconfigured port is always returned as zero.

1 means that the controller is the Controller in Charge on this port. If the SYC (system controller) switch on the IEEE-488 interface board is set true, the controller will be in the CIC state when the Operating System program is loaded (upon power-up or after the RESTART button is pressed).

2 means that the 1722A has been addressed as a listener on this port. This implies that an INPUT statement should be executed to read a command from the Controller in Charge of the interface on this port.

3 means that the 1722A has been addressed as a talker on this port. This implies that a PRINT statement should be executed to send data to the Controller in Charge of the interface on this port.

The DCL, GET, and SPMS status bits are set and reset automatically. The GET, DCL, and SPMS status bits for a particular port will be reset to zero when

1. The PORTSTATUS() function for the port is used, or when
2. A RESUME statement is executed which terminates the ON PORT interrupt handler for the port.

PORTSTATUS()
Function

IEEE-488

Example

The following code segment is an example of the use of the PORTSTATUS() function. This code can be used either to determine the cause of an ON PORT interrupt or simply to periodically poll the state of the IEEE-488 bus.

```
1000 ST% = PORTSTATUS(TP%)                    ! read port's status
1010 SA% = ST% AND 3%                         ! isolate controller state
1020 TR% = ST% AND 16%                        ! triggered?
1030 CL% = ST% AND 8%                         ! received DCL or SDC?
1040 PL% = ST% AND 4%                         ! was SRQ acknowledged?
1050 IF TR% THEN ...                          ! action for GET command
1060 IF CL% THEN ...                          ! action for clear command
1070 IF PL% THEN ...                          ! action for SRQ acknowledge
1080 ON SA% + 1% GOTO 1090, 110, 1110, 1120
1090                                          ! code for idle port
...
1100                                          ! code for Controller
...
1110                                          ! code for Listener
...
1120                                          ! code for Talker
...
```

Usage

PPL (port number)

Description

PPL (Parallel Poll) performs a Parallel Poll of a specified instrument bus port and returns an integer result.

- The result is an integer value between 0 and 255.
- Refer to Appendix I, ASCII/IEEE-488 Bus Codes, for a chart of binary patterns and decimal numbers. To use this chart, read the value returned by PPL in the decimal column and read the binary pattern in the binary column.
- The correspondence between the IEEE-488 DIO lines and binary bit numbers is as follows:

DIO Line	Bit Number	Numeric Weight
DIO1	0	1
DIO2	1	2
DIO3	2	4
DIO4	3	8
DIO5	4	16
DIO6	5	32
DIO7	6	64
DIO8	7	128

Example

The following example assigns the results of a parallel poll of port 0 to the integer variable Y%.

```
4710 Y% = PPL(0%)
```

An AND may be used to test individual bits of the parallel poll result. The following example performs a parallel poll of port 1. If the DIO8 line were asserted true, the statement(s) following THEN would be executed.

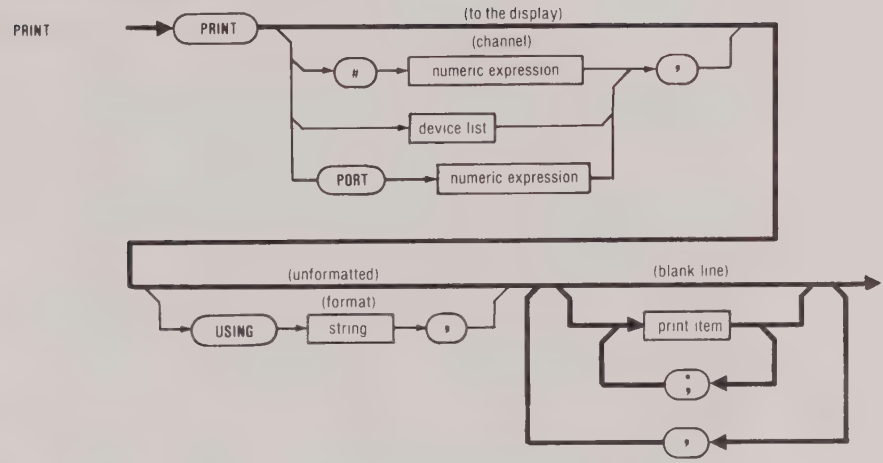
```
8460 IF PPL(1%) AND 128% THEN ...
```

Statement

Usage

PRINT expression

Syntax Diagram



Description

The PRINT statement sends string, integer, or floating-point data to the display, an opened Channel or the IEEE-488 Bus.

- PRINT sends data to the display unless otherwise specified in the statement.
- A single PRINT statement can send the results of more than one expression, including single or multiple constants or variables.
- Successive data items following PRINT must be separated by a comma or semicolon, to format the display.
- Unless otherwise specified, the data sent by a PRINT statement will be followed by a pair of ASCII control characters (carriage return and line feed).
- A comma or semicolon may be used to terminate a PRINT statement and suppress the Carriage Return and Line Feed Characters sent at end-of line. This will cause a succeeding PRINT statement to display data on the same line.

PRINT
Statement

- Data items separated by commas are displayed in 16-character print fields with up to five print fields per line.

- If a data item is longer than one print field the next data item falls into the next empty print field even if the next print field is a succeeding 80 character line.

- A numeric data item is displayed with a leading space or sign, and a trailing space. It is printed out to seven significant digits. A value from .1 to 9999999 inclusive is printed out directly. A number less than .1 is printed out without E notation if all of its significant digits can be printed. All other values are printed in E notation (+0.dxxxxxxE+yyy), where d is a non-zero digit, x is any digit, yyy is the exponent, and trailing zeros are dropped.

- Data items separated by semicolons are displayed side-by-side with no added spaces (other than those generated during numeric to decimal ASCII conversions).

- PRINT returns the cursor to the left-hand end of the next display line if the last data item is not followed by a comma or semicolon, or if no data items are specified.

- If the cursor is on the bottom line, PRINT scrolls the display upwards one line, unless the PRINT command is a cursor positioning command, or the display is in Page Mode.

Examples

The following examples illustrate some common uses of the PRINT statement:

STATEMENT	RESULTING DISPLAY ^	
PRINT,,,	(48 spaces)	
PRINT 4,5	4 (15 spaces)	5 (1 space) ^
PRINT 5;-6;	5-6 ^	
A$="Hz" PRINT "K";A$;	KHz^	

The following program example illustrates typical usage of PRINT statements to display readings taken from a voltmeter. The "," is used to format the displays in columns, and the ";" to print out the units together with the values. The comma at the end of line 1090 ensures that "** FAILED **" will be printed on the same line as test results that were found out of limits. Line 1120 ensures that when the test does not fail, any further data will not be displayed on the same line. The ";" at the end of lines 1100 and 1110 suppresses the carriage return and line feed after "** FAILED **" is displayed. Therefore, line 1120 has the same effect whether the test passes or fails.

```
1000 REM -- Display Readings In Volts and Display Limits
1010                                    ! R is value read by voltmeter
1020                                    ! LL is lower limit
1030                                    ! UL is upper limit
1040                                    ! Print ** FAILED ** if reading
1050                                    ! is outside tolerances
1060                                    ! Heading:
1070 PRINT "LOWER LIMIT", "READING", "UPPER LIMIT"
1080 PRINT                              !Leave blank line
1090 PRINT LL; 'V', R; 'V', UL; 'V',
1100 IF LL ) R THEN PRINT '** FAILED **'; !Below tolerance
1110 IF UL ( R THEN PRINT '** FAILED **'; !Above tolerance
1120 PRINT                              !Next line
```

Assume LL has the value 2.9 and UL has the value 3.1. Then for various values of R, the display would appear as follows:

```
LOWER LIMIT     READING         UPPER LIMIT
2.9V            2.87V           3.1V            ** FAILED **
LOWER LIMIT     READING         UPPER LIMIT
2.9V            3.04V           3.1V
LOWER LIMIT     READING         UPPER LIMIT
2.9V            3.103V          3.1V            ** FAILED **
```

The following example shows PRINT as an Immediate Mode command. The result of the command is a single line of values as follows:

```
PRINT 2; EXP(2), 4; EXP(4), 6; EXP(6), 8; EXP(8)
 2  7.389056  4  54.59815  6  403.4288  8  2980.958
```

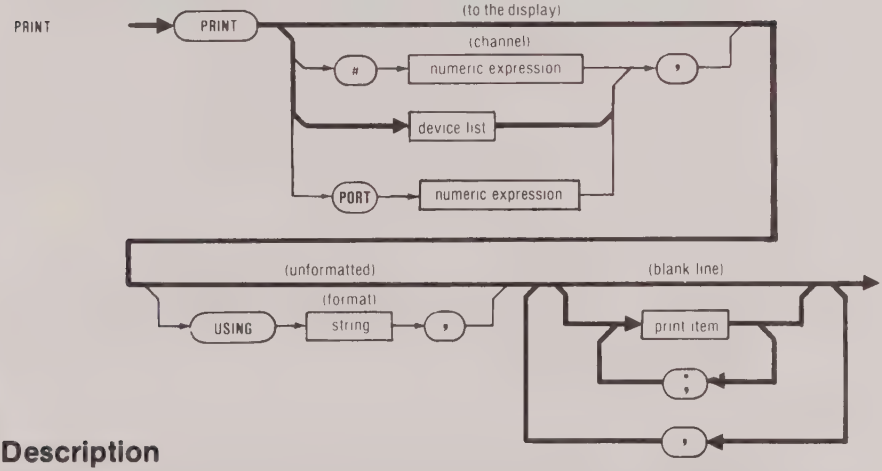
NOTE

EXP(n) is the result of raising e (the natural log base) to the power n.

Usage

PRINT [USING] @ [device list], expression
PRINT [USING] PORT p%, expression

Syntax Diagram



Description

PRINT and PRINT USING are used to output data to designated listeners. The instruments which are to receive output data are specified in a device list.

- A PRINT or PRINT USING command which is followed by a device list addresses the specified devices as listeners.
- All other devices are commanded to unlisten.
- When only an @ character follows PRINT or PRINT USING, with no device numbers, then no listen, unlisten, talk, or untalk commands are sent. Data is sent to the last bus address. This will increase the speed of data output.
- One or both instrument ports may be represented in the device list.
- Output data is sent character by character.

- Characters are sent exactly as a normal PRINT or PRINT USING statement, except for the use of the comma and semicolon to format the output.

- A comma following a data item in the output list indicates the EOI Bus line is to be set simultaneously with the last character of that item. It does not indicate tabulation to 16 character columns by sending extra spaces.

- A Carriage Return and Line Feed, with the EOI Bus line set simultaneously with the Line Feed, follows the last data item in a PRINT list when it is not terminated with either a comma or a semicolon.

- Array subranges may be included in the output list.

- A comma or semicolon following a subrange specification is equivalent to listing each element of the array subrange followed by the comma or semicolon.

Examples

The following example sends the characters "F0RA3" to the DVM if the value F1 is 0 and R1 is 3. Note the use of the mask "#" to eliminate the leading and trailing space that would normally be printed with F1 and R1. After the 3, a Carriage Return and a Line Feed (with EOI set true) is sent.

```
1700    REM -- Program DVM on Port 1, Address 4
1710    ! Send function and range:  F1 and R1
1720    PRINT @ 104%, USING "#", "F", F1, "RA", R1
```

The following Immediate Mode PRINT statement addresses instruments on port 0 at device 2 and at device 4 with secondary address 6 as listeners. It then sends out the contents of strings A$(3) through A$(6) and sets EOI true with the last byte of each string.

```
PRINT @ 2 @ 4:6, A$(3%..6%),
```

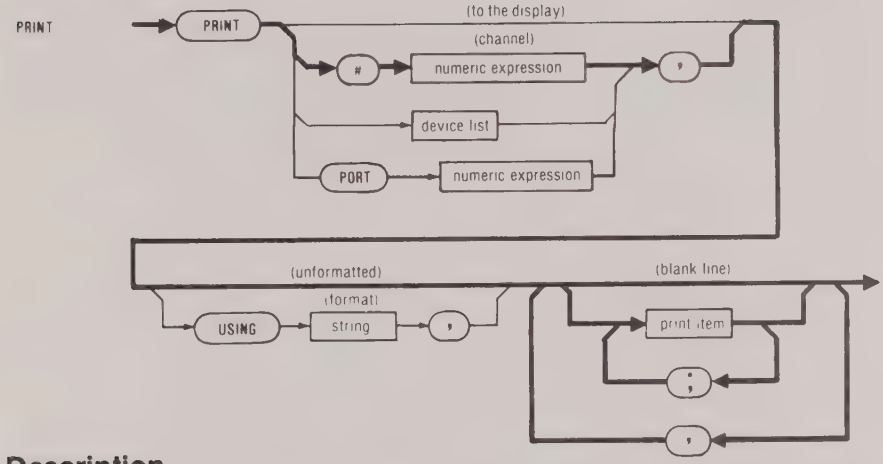
The following example selects the instrument at address 2 as a listener at line 100. The indicated data is then sent and the instrument remains a listener. Since there is no other instrument bus activity before line 200, the characters "R1?" will be sent to the same instrument.

```
100     PRINT @ 2, "F153R0?"          ! Set up instrument
110                                   !
120                                   ! other non-instrument statements
130                                   !
200     PRINT @, "R1?"                ! Change range of instrument
210                                   ! other statements
220                                   !
```

Usage

PRINT expression

Syntax Diagram



Description

PRINT is described here for output to a file through a previously opened channel. PRINT may also be used for direct display output, or for output to instruments on the IEEE-488-1980 bus. These applications of PRINT are described in other sections in this manual.

- The numeric expression following PRINT selects a previously opened channel. See the OPEN statement described in this section.
- The items to be printed are listed, separated by either a comma or a semicolon.
- The items to be printed may include integer, floating point, and string expressions, as well as subranges of arrays and virtual arrays. Refer to the section on Data, Data Types, and Expressions, for a discussion of array subranges.
- When the channel is attached to an RS-232 port, the End-of-File character (as defined by the SET Utility program) will be substituted for any ⟨CTRL⟩/Z, CHR$(26), characters.

- The USING option may be specified for formatted output. Refer to the PRINT USING section for details.

Examples

The following examples illustrate some of the implications of specifying an array subrange in a PRINT statement. For a one-dimensional array, the designation A(m..n) is equivalent to the list of elements A(m) through A(n). Formatting may be as follows:

1. With semicolons:

   PRINT A(m..n);

   is equivalent to:

   PRINT A(m); A(mb1); . . . ; A(n);

2. With commas:

   PRINT A(m..n),

   is equivalent to:

   PRINT A(m), A(mb1), . . . , A(n),

3. With neither commas nor semicolons:

   PRINT A(m..n)

   is equivalent to:

   PRINT A(m)

   PRINT A(m+1)

   .
   .
   .

   PRINT A(n)

The following example illustrates the sequence of output of a two-dimensional array subrange. As with one-dimensional output, the formatting may be with semicolons, commas, or neither. This sequence is often described as "row-major ", i.e., the first subscript (the row) changes slowest.

```
PRINT A(m..n, r..s)

is equivalent to:

PRINT A(m, r)
PRINT A(m, rbl)
 .
 .
 .
 PRINT A(m, s)
PRINT A(mbl, r)
PRINT A(mbl, rbl)
 .
 .
 .
 PRINT A(mbl, s)
 .
 .
 .
 PRINT A(n, r)
PRINT A(n, rbl)
 .
 .
 .
 PRINT A(n, s)
```

In the following three examples, NV(I) is a 11-element array used to store 10 expected nominal values for readings to be taken by an instrument. (Element zero is not used.) The 10 actual readings are in the array R(I). The examples illustrate different options for the printout format.

PRINT #n
Statement

This first example produces three columns of data. Each element of data is printed out in default format. Note that this results in some items that are difficult to read, especially in the column marked %ERROR. The comma between the elements of the PRINT statement (line 1050) produces the columns.

```
10    OPEN "KB1:" AS NEW FILE 1
20    ! Other statements
30    !
200   GOSUB 1000
210   ! Other statements
220   !
250   CLOSE 1 \ STOP
1000  REM -- Subroutine: Output Readings To Printer
1010  PRINT #1                                              ! Blank line
1020  PRINT #1, 'NOMINAL VALUE', READING', '% ERROR'        ! Heading
1030  PRINT #1,                                             ! Blank line
1040  FOR I = 1 TO 10                                       ! Print results:
1050  PRINT #1, NV(I), R(I), (R(I) - NV(I)) / NV(I) * 100
1060  NEXT I
1070  RETURN
```

Results:

```
NOMINAL VALUE    READING          %ERROR

 2.222            2.19527         -1.20298
 4.444            4.361702        -1.851896
 6.666            6.758841         1.392759
 8.888            8.891673         0.4133037E-01
-2.222           -2.246359        -1.096246
-4.444           -4.50368          1.342932
-6.666           -6.545657        -1.805325
-8.888           -8.715492        -1.940907
 100              98.70009        -1.299913
-100             -99.38668         0.6133201
```

This second example is an improvement on the previous one. The USING option of the PRINT statement formats the output as described in the entry for the PRINT USING STATEMENT. The semicolons (;) in line 1050 separate numeric data elements from units designators (in this case, V for voltage). This results in a more readable format. Note the asterisks (**) in the last line of the % ERROR column, this results when a numeric value does not fit within the designated mask. The actual value in default format follows the question mark.

```
10   OPEN "KB1:" AS NEW FILE 1
.
.
250  CLOSE 1 \ STOP
1000 REM -- Subroutine: Output Readings To Printer
1005 ! Linearity Values:
1010 PRINT #1, USING 'S##.###', 'LIN:       '; NV(1..4);
1015 PRINT #1                                              ! Blank line
1020 PRINT #1, USING 'S##.###', 'RDGS:      '; R(1..4);    ! Print readings
1030 PRINT #1                                              ! Blank line
1040 PRINT #1                                              ! Blank line
1045 ! Negative Linearity:
1050 PRINT #1, USING 'S##.###', 'NEG LIN:   '; NV(5..8);
1055 PRINT #1                                              ! Blank line
1057 ! Negative readings:
1060 PRINT #1, USING 'S##.###', 'NEG RDGS:  '; R(5..8);
1065 PRINT #1                                              ! Blank line
1070 RETURN
```

Results:

```
 NOMINAL VALUE    READING          % ERROR

     2.222V         2.195V          -1.2%
     4.444V         4.362V          -1.9%
     6.666V         6.759V           1.4%
     8.888V         8.892V           0.0%
    -2.222V        -2.246V           1.1%
    -4.444V        -4.504V           1.3%
    -6.666V        -6.546V          -1.8%
    -8.888V        -8.715V          -1.9%
   100.000V        98.700V          -1.3%
  -100.000V         0.000V          *****
```

This third example illustrates the use of a subrange specification to print selected elements of an array. Elements of arrays NV and R, the nominal values and readings, are each printed one after the other, using a subrange specification. The string mask following USING is organized to space the elements. The semicolons used in the PRINT statement allow many elements to be printed on one line. Note the use of the PRINT statement at lines 1015, 1030, 1040, 1055, and 1065 to force a Carriage Return and Line Feed.

```
10    OPEN "KB1:" AS NEW FILE 1
20    ! Other statements
30    !
200   GOSUB 1000
210   ! Other statements
220   !
250   CLOSE 1 \ STOP
1000  REM -- Subroutine: Output Readings To Printer
1010  PRINT #1                                          ! Blank line
1020  PRINT #1, 'NOMINAL VALUE', READING', '% ERROR'    ! Heading
1030  PRINT #1,                                         ! Blank line
1040  FOR I = 1 TO 10                                   ! Print results:
1050  PRINT #1, NV(I), R(I), (R(I) - NV(I)) / NV(I) * 100
1060  NEXT I
1070  RETURN
```

PRINT #n
Statement

Results:

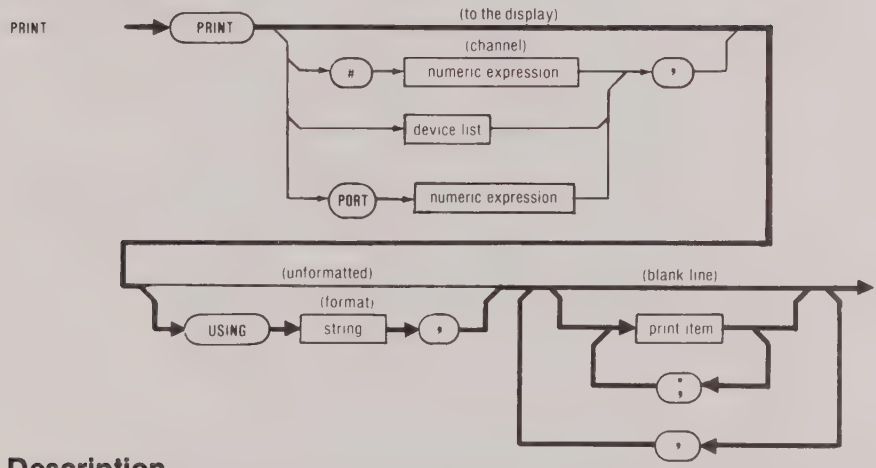
```
LIN:          2.222   4.444   6.666   8.888
RDGS:         2.195   4.362   6.759   8.892
NEG LIN:     -2.222  -4.444  -6.666  -8.888
NEG RDGS:    -2.246  -4.504  -6.546  -8.715
```

PRINT USING Statement

Usage:

PRINT USING format ; expression

Syntax Diagram:



Description

The optional USING specification of the Print Statement designates a specific format for data output.

- The string mask following USING specifies the format for all numeric information to be output. Table 1 presents the string mask characters.
- The string mask may be predefined in a string variable, or must be enclosed in quotes.
- Leading zeros are not output, except that one leading zero is output whenever there are no significant digits to the left of the decimal.
- A space is output in the position reserved for a sign when the sign is positive.
- Data is rounded at the least significant digit specified in mask when it contains additional digits.
- The string mask has no effect on string output.

- If the number cannot be represented using the format indicated, the string printed will be a field of asterisk (*) characters filling the entire number field (i.e. the asterisk string will take the same space allocated by the format string for the number).

- Format overflow may be caused by any of the following conditions:

  1. A negative number must be printed but no sign position (S or P in the format string) has been specified.

  2. The field width to the left of the radix (decimal) point is too small to hold the number. This will never occur with the exponential format.

  3. A two-digit exponent field has been specified, but the actual exponent magnitude is greater than 99.

Table 1

CHARACTER	MEANING
C or c	Indicates that the comma (,) is to be used for the radix point. If the "," specification is used to comma-separate the output with the "C" format specification, the period (.) is used as the digit separator. This conforms to the European numeric formatting conventions.
P or p	Indicates that the sign should always be printed. A "+" will be printed for positive numbers.
S or s	Reserves a position for a sign symbol (+ or –). When positive, a space is output. When negative, a "–" character is used. Only one "S" may be used in the mask, and it must be the first character.
#	Reserves a position for a digit or a space.
.	Reserves a position for a decimal point. Only one "." may be used in the mask. When a comma is included in the mask, it must precede the decimal point.
,	Indicates that a comma is to be placed every three digits to the left of the decimal point. When a decimal point is included in the mask, it must follow the comma.
^^^^	Reserves four positions for an exponent. When used, these four carets must be the last characters in the mask. The output will be left justified, using the number of digits and the decimal position defined in the mask, followed by "E+mn" or E-mn". The number output for "mn" is the number of places the decimal point must be moved to restore the original form of the number (" for right, - for left).
^^^^^	As above, except "^^^^ " reserves five positions for an exponent. The exponent is printed as a 3-digit field.

PRINT USING
Statement

Examples

The following example illustrates PRINT USING processing various number forms through the same string mask:

```
PRINT USING "$****.**", -.9, 99, 9999, 99999, "Nine"
```

```
-0.90          99.00          9,999.00          *********          Nine
```

This next example illustrates a PRINT USING statement in a program that reads numeric data and then displays it in two different special purpose fromats using masks stored in string variables.

```
100     A$ = "$****,.**" \ B$ = ".###"
110     FOR I = 1 TO 4
120        READ A, B
130        PRINT A; B; TAB (20); "$";
140        PRINT USING A$, A,
150        PRINT USING B$, B
160     NEXT I
170     DATA 9999.99, .888, .03, .00368, .00002230, -.12
180     DATA 222222, 7777
```

```
Ready
RUN
9999.99 0.888                  $9,999.99              .888
0.03 0.00368                   $    0.03              .004
0.0000223 -0.12                $    0.00              ****
222222 7777                    ********              ****
```

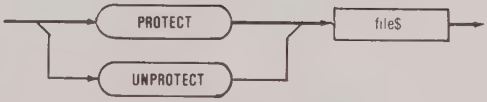
The following example illustrates the effects of exponential notation as output by PRINT USING:

```
PRINT USING "$##.###^^^^", .0004, 9E-14, -4356
40.000E-05        90.000E-15        -43.560E+02
```

Usage

PROTECT {file$}

Syntax Diagram



Description

The PROTECT statement assigns a protection code to a specified file to prevent it from being deleted (e.g., via KILL or OPEN AS NEW FILE).

- The "+" protection code is visible using the DIR or EDIR statements to examine the directory containing the specified file.
- Attempting to delete a protected file will result in an error.

Example

The instruction

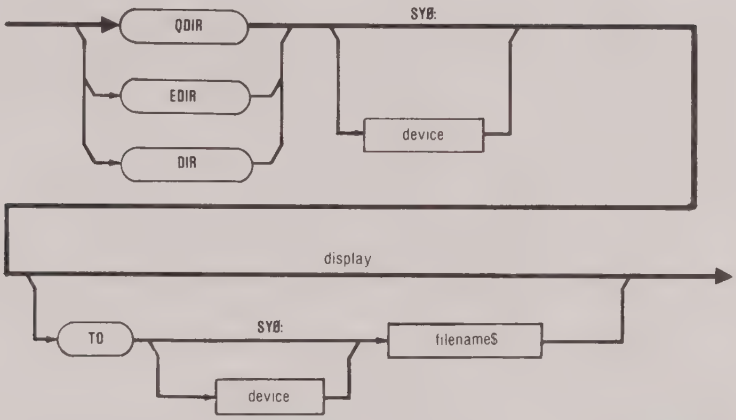
```
10255 PROTECT "RPFILE.TMP"
```

assigns a "+" protection code to the file RPFILE.TMP to protect it from deletion.

Usage

QDIR [device$] [TO filename$]

Syntax Diagram



Description

The QDIR statement prints a short (or "quick") directory listing in which only the filenames (not the sizes, protection codes, or file creation dates) are given.

- Device$ is the optional name of the device for which a directory listing is desired (which is "SY0:" if omitted).
- Filename$ is the optional name of a file to which the directory listing is to be sent (which is "KB0:" if omitted).

The output format is:

file1 file2 file3 file4 file5 file6
file7 ...

Example

The statement

QDIR

displays the directory of SY0: on KB0: (the display).

Usage

RAD$(integer%, base%)

Description

The RAD$ function converts a numeric value to a printable string of digits in a non-decimal base. It returns a string (without leading zeroes or blank padding) corresponding to integer% as a number using the given base base%.

- "digits" greater than 9 in the output string will be returned as "A" through "Z".
- Any floating-point arguments will be truncated to an integer, which may cause an arithmetic overflow (error 603).
- The value of intger% is limited to the range of integers.
- The sign of base% determines whether or not a "–" character will precede the ASCII representation of the number. If base% is positive, no "–" character will precede a negative integer%. If base% is negative, a "–" character will precede the representation for a negative integer%.
- An error 516 (ill-formed expression) will be reported if the absolute value of base% is less than or equal to 1.
- If the value of base% is greater than 36, strange characters may appear in the string produced by RAD$().

Example

```
PRINT RAD$(-16%, 16%), RAD$(-16%, -16%)
```

would print

```
FFF0            -10
```

The statement

```
PRINT RAD$(32%, 16%), RAD$(32%, 8%), RAD$(32%, 2%)
```

would print

```
20              40              100000
```

Usage

RANDOMIZE

Syntax Diagram

Description

The RANDOMIZE statement generates a random seed number from the system clock that is used internally by the RND function. The RND function normally creates a repeatable psuedo-random series of numbers. When the RANDOMIZE statement is used, the RND function creates a truely random non-repeatable sequence of numbers.

- RANDOMIZE may be used in the immediate mode to influence random numbers which are also used in the immediate mode.

- It is recommended that RANDOMIZE be used only once in a program. Repeated RANDOMIZE statements may cause the RND function to produce non-random results.

Example

Following is an example of how the RANDOMIZE statement works in a program. The example begins with a program WITHOUT a randomize statement:

```
100 FOR I=1 TO 5                    ! For 5 times
110     PRINT RND;                  !     Print a random number
120 NEXT I
```

Running this program any number of times will each produce exactly the same results, because without the RANDOMIZE statement, the RND function prints a repeatable sequence (which is reset to the beginning by the RUN statement):

```
 0.2874767    0.1992549    0.1633877    0.3702594E-01    0.2543282E-01
```

Adding the line

```
90 RANDOMIZE                        ! Create a random seed
```

to the program above creates a new random set of numbers each time that the program is run.

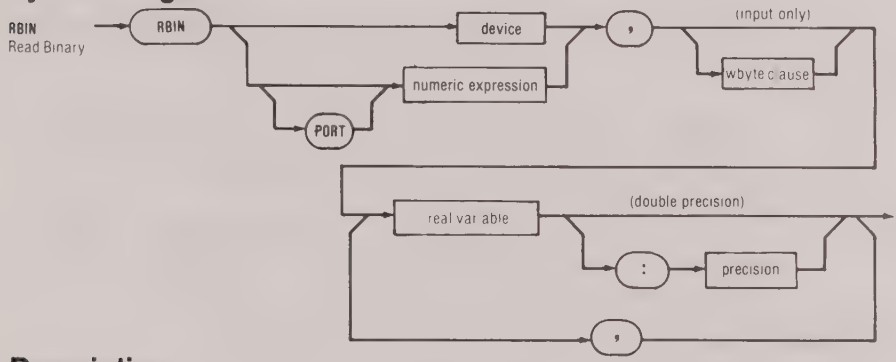
Statement

IEEE-488

Usage

RBIN [device], [wbyte clause] [variable or array subrange]
RBIN [port numeric expression], [wbyte clause] [variable or array subrange]

Syntax Diagram



Description

The RBIN (Read BINary) statement receives single- and double-precision data in IEEE standard floating-point format from IEEE-488 bus instruments.

- The specified instrument device number is addressed as a talker.
- Instrument addressing is skipped when the @ character follows RBIN without a device specified.
- When the statement includes a WBYTE clause, data specified by the WBYTE clause is then sent to the specified port. See the WBYTE statement.
- The WBYTE clause selects a subrange of an integer array for output. This may contain addressing, bus messages, and device-dependent data.
- The WBYTE clause does not need to be sent to the same instrument port from which input is being taken.
- Make sure that the WBYTE output does not unaddress the instrument as a talker if it is sent to the port from which input is read.
- A single floating-point value is then received from the specified instrument. The value must be in the selected format.

RBIN
Statement

IEEE-488

- :4 specifies a four-byte single-precision number.
- :8 specifies an eight-byte double-precision number.
- The default format is eight-byte double-precision.
- The variable or array subrange to receive the data must be floating-point variable type.
- Single-precision data is converted to double-precision (internal format) and assigned to the floating-point variable. No conversion is necessary for data received in double-precision form.
- If additional data is to be received, and a WBYTE clause is specified, the data specified by the WBYTE clause is sent again prior to each reading.

Example

The following example addresses device 5 on port 1 as a talker, and then reads six single-precision floating-point values from port 1. The single-precision data will be converted to double-precision and assigned to elements 0 through 4 of array A and to variable C.

```
3650    RBIN @ 105%, A(0..4):4, C:4
```

The following example reads one double-precision value from the port 0. It assigns the value to variable B. Note that no device addressing is performed since a "port" address is specified.

```
10940   RBIN PORT 0, B:8
```

The following example reads 30 double-precision (8-byte) values from device 20 on port 0. Prior to reading each value from port 0, the WBYTE data (C%(0..5)) will be sent to port 0. The values are assigned to matrix B in the following order:

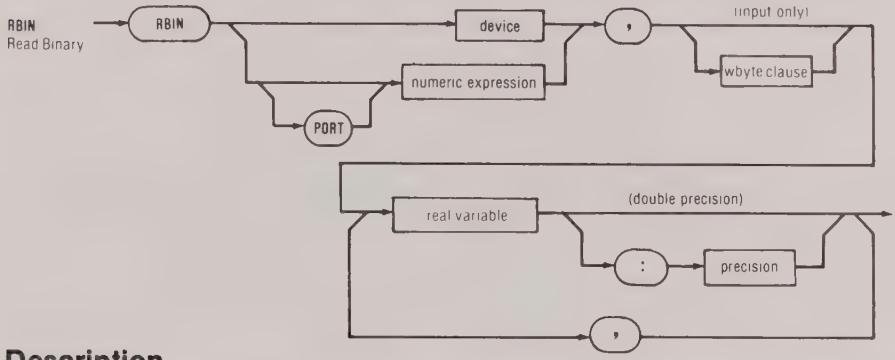
B(0,0)
B(0,1)
.
.
B(0,5)
B(1,0)
.
.
B(4,4)
B(4,5)

```
4720    RBIN @ 20%, (WBYTE PORT 0%, C%(0%..5%)) B(0%..4%, 0%..5%)
```

Usage

RBIN {device}, {wbyte clause} {real variable list}
RBIN @, {wbyte clause} {real variable list}
RBIN {port expression}, {wbyte clause} {real variable list}

Syntax Diagram



Description

The RBIN WBYTE (Read BINary WBYTE) statement addresses the specified device(s) as talker(s), sends the data specified by the WBYTE clause, then receives single- and double- precision data in IEEE standard floating-point format from IEEE-488 bus instruments.

- The specified instrument device number is addressed as a talker.
- Instrument addressing is skipped when the @ character follows RBIN without a device specified.
- The WBYTE clause sends data specified by the WBYTE clause to the specified port. See the WBYTE statement.
- The WBYTE clause selects a subrange of an integer array for output. This may contain addressing, bus messages, and device-dependent data.
- The WBYTE clause does not need to be sent to the same instrument port from which input is being taken.
- Make sure that the WBYTE output does not unaddress the instrument as a talker if it is sent to the port from which input is read.

RBIN WBYTE
Statement

IEEE-488

- A single floating-point value is then received from the specified instrument. The value must be in the selected format.
- :4 specifies a four-byte single-precision number.
- :8 specifies an eight-byte double-precision number.
- The default format is eight-byte double-precision.
- The variable or array subrange to receive the data must be floating-point variable type.
- Single-precision data is converted to double-precision (internal format) and assigned to the floating-point variable. No conversion is necessary for data received in double- precision form.
- If additional data is to be received, and a WBYTE clause is specified, the data specified by the WBYTE clause is sent again prior to each reading.

Example

The following example reads 30 double-precision (8-byte) values from device 20 on port 0. Prior to reading each value from port 0, the WBYTE data (C%(0..5)) will be sent to port 0. The values are assigned to matrix B in the following order:

B(0,0)
B(0,1)
.
.
B(0,5)
B(1,0)
.
.
B(4,4)
B(4,5)

```
4720    RBIN @ 20%, (WBYTE PORT 0%, C%(0%..5%)) B(0%..4%,0%..5%)
```

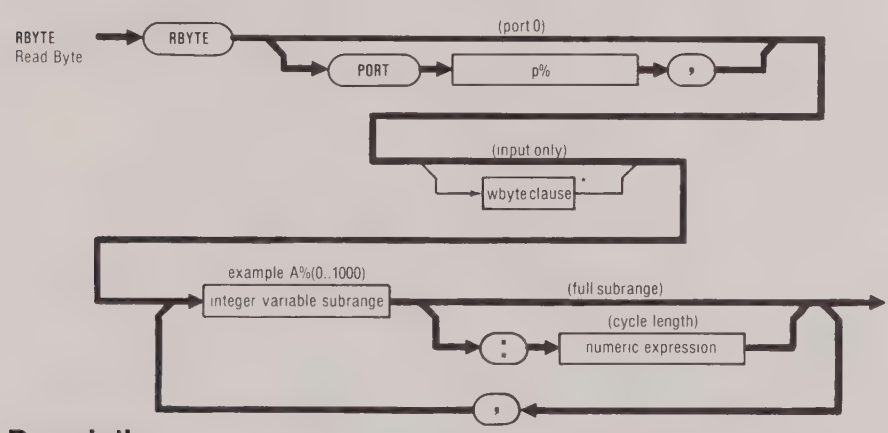
Remarks

Refer to the RBIN statment.

Usage

RBYTE [PORT numeric expression,] [integer variable subrange]

Syntax Diagram



Description

The RBYTE (Read BYTE) statement reads a fixed-length block of arbitrary binary data bytes from an instrument. The data from the instrument is placed in a specified integer variable array.

- The array must have only a single dimension (subscript).
- The array must be integer type.
- A virtual array may not be used.
- The ATN Bus line is first set false.
- Data is then read into the integer array elements as follows (bit 0 is low-order):

BITS	CONTENTS
0-7	8-bit data byte
8	Set to 1 if EOI was asserted by the talking device with this data byte
9-15	Set to zero

RBYTE
Statement

IEEE-488

- A data byte sent with EOI asserted will then have the following characteristics:

  1. Its numeric value will be greater than 255.

  2. An AND with the 2^8 bit will produce a non-zero value. For example:

```
IF A%(5%) AND 256% THEN ...(EOI is true)
```

- RBYTE performs no device addressing. The WBYTE clause may be used to address an instrument as a talker.

- Cycle length is described in the RBYTE WBYTE statement.

Example

The following example reads one data byte from port 0 into element 0 of the integer array A%. The instrument has already been addressed to talk.

```
2550    RBYTE PORT 0%, A%(0%)
```

The following example reads fifteen data bytes from port 0 (the default port) into array A%, elements 0 through 14. Then seven data bytes are read, also from port 0, into elements 12 through 18 of the array C%.

```
5890    RBYTE A%(0%..14%), C%(12%..18%)
```

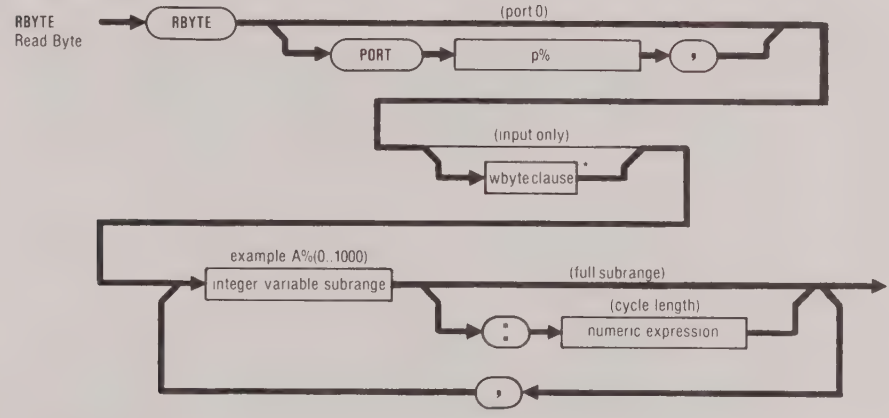
The following example uses RBYTE to take 100 readings, each three bytes long. To insure that the readings are correct, each third byte is tested to see that EOI was set.

```
1200 REM -- Subroutine: Get Instrument Reading
1210 ! other statements to set up instrument as talker, etc.
1220 !
1250 ! Now get readings
1260 RBYTE PORT P%, R(1%..300%)
1270 ! Insure that EOI is sent every third byte
1280 EF% = 0                    ! Error Flag = False
1290 FOR I% = 3% TO 300% STEP 3%! Error Flag = True?
1300 IF NOT (R(I%) AND 256%) THEN I% = 300% \ EF% = -1
1310 NEXT I%
1320 IF EF% GOTO 1400           ! Error exit
1330 RETURN
```

Usage

RBYTE [PORT p%,]] [WBYTE] [integer array subrange]

Syntax Diagram



Description

The WBYTE clause added to the RBYTE statement provides a means of sending commands or data to a port (via the WBYTE clause) prior to reading the data as specified by RBYTE.

- RBYTE WBYTE can be used for an instrument that requires an explicit trigger for each reading.
- The WBYTE data is sent prior to each RBYTE cycle.
- The cycle length is specified by the value of the expression following the colon ":".
- If no cycle length is specified, it is assumed to be the length of the array subrange received.
- Only one WBYTE subrange may be specified.
- Each array must have only a single dimension (subscript).
- Each array must be of the integer data type.
- No virtual arrays may be used.

RBIN WBYTE Statement

IEEE-488

- The total number of bytes read must be evenly divisible by the number of bytes per cycle. An RBYTE data specification A%(1..N):M reads a total of N bytes in N/M cycles of M bytes each.

Example

The following example transmits the WBYTE data (elements 0 through 4 of array A%) on port 1 prior to reading elements 0 through 5 of array B% (from port 0), and prior to reading elements 0 through 11 of array C% (also from port 0). Each array subrange constitutes an RBYTE cycle.

```
6840    RBYTE (WBYTE PORT 1%, A%(0%..4%)) B%(0%..5%), C%(0%..11%)
```

The following example receives data read from port 0 (elements 0 through 59 of array A%) in 20 cycles of 3 bytes each. The WBYTE data is sent to port 0 prior to each RBYTE cycle. This statement will send an explicit trigger for each reading if the array Z%(0..5) is assigned as the D% array was in the third example of the WBYTE description.

```
5380    RBYTE (WBYTE Z%(0%..5%)) A%(0%..59%):3%
```

The sequence of this statement is:

1. Send Z%(0..5)
2. Read A%(0..2) . . . (first 3 elements)
3. Send Z%(0..5)
4. Read A%(3..5) . . . (second 3 elements)
5. Send Z%(0..5)

   . . .

   . . .

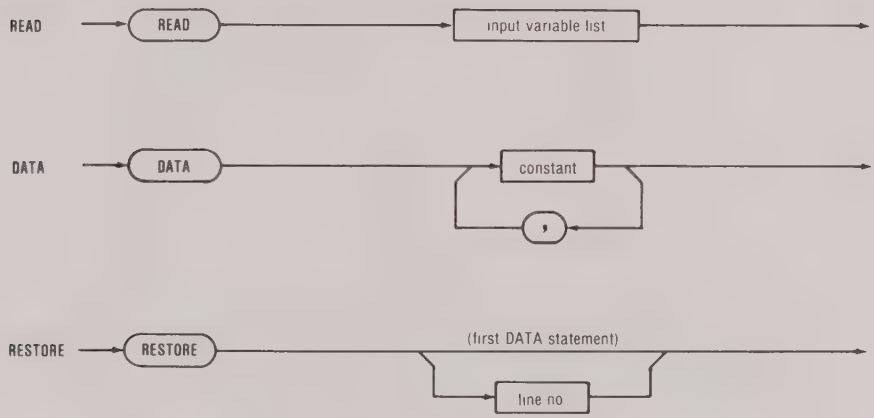
39. Send Z%(0...5)
40. Read A%(57..59) . . . (last 3 elements)

READ, DATA, and RESTORE Statement

Usage

READ data
DATA data list
RESTORE linenumber

Syntax Diagram



Description

The READ, DATA, and RESTORE statements work together as follows:

- The DATA statement defines a sequence of data items to be processed by the READ statement. The data items within a single DATA statement are separated by commas.
- The READ statement assigns data values to a series of one or more variables.
- An array subrange may also be used with the READ statement.

  READ A(0..5)

- The READ statement assigns the next available data items in sequence to the variables referenced.
- The DATA data types must match the corresponding READ variable types (strings for string variables, etc.).

READ, DATA, and RESTORE Statement

- The DATA statement may occur before or after the READ statement.
- The DATA statement must be the last or only statement in the program line. The line may contain no trailing remarks. Everything following DATA is considered to be data.
- Legal data items are: quoted or unquoted strings or numeric constants.
- An unquoted string must not begin with a quote and must not contain commas.
- Leading spaces are ignored unless within a quoted string field.
- Numeric constants may not be quoted.
- A data pointer tracks which data items have been read.
- A DATA statement may contain more items than a subsequent READ statement contains variables.
- A second READ statement may continue reading data (assigning data items to its variables) at the point the first READ statement stopped reading.
- The RESTORE statement resets the pointer to the first data item of the first DATA statement in the program so that the items may be read again.
- If RESTORE specifies a line number, the pointer resets to the first data item of the first DATA statement in or after that program line.
- The RESTORE statement may be executed before all data items have been read from a DATA statement.
- It is not necessary to RESTORE the DATA items if they will be read only once in the program.

Examples

This example illustrates the use of a FOR - NEXT loop to read a list of data items. Each item is a floating-point number, so only one READ statement is necessary. To save each of the data items, use arrays and change line 110 to: READ A(I%). The RESTORE statement in this example would be useful only if the data values were to be used again in the program.

```
10 DATA 1.1, 2.2, 3.3, 4.4, 5.5
20 ! Other statements
100 FOR I%=1% TO 5%
110 READ A
120 PRINT A
130 NEXT I%
140 RESTORE
```

Running this program gives results as follows:

```
1.1
2.2
3.3
4.4
5.5
```

The following example illustrates multiple READ statements and a selective RESTORE statement. Note the double quotes used in the first DATA statement. The double quote is used to allow commas to be inserted in the string data. Note that if the quotes were omitted from line 10, A$ would be TEST FOR HIGH, and line 120 would give an error since the next data element would be LOW, which is not a number. Also note line 410 which resets the data pointer to allow reading the numeric values of line 20. The string data on line 10 is used only once.

```
10 DATA 'TEST FOR HIGH, LOW, AND MEAN VALUES.'
20 DATA 10, 7.5, 9
30 READ A$ \ PRINT A$                          ! Print out heading
100 REM -- Continuously Make Checks
110 ! Check for readings higher than upper limit
120 READ UL                                    ! Get the upper limit
130 ! Other statements
210 ! Check for readings lower than lower limit
220 READ LL                                    ! Get the lower limit
230 ! Other statements
310 ! Compute mean and compare to expected value
320 READ EV                                    ! Get the expected value
330 ! Other statements
400 REM -- Reset instruments and Prepare for Next Test
410 RESTORE 20                                 ! Reset data pointer to UL
420 ! Other statements
500 GOTO 110
```

The program segment that follows causes error 804 (bad DATA format) when statement 110 is executed, since the "!" character is not a legal part of a floating-point number.

```
100  DATA 1,3,4      ! Configuration data
110  READ A,B,C
```

Description

Relational operators compare numeric values or character strings. A relational expression returns an integer Boolean result of 0 for false, and −1 for true. The structure of a statement determines whether the = operator is used for a relational comparison or an assignment.

The relational operators are:

= equal

< less than

> greater than

<> not equal

<= less than or equal

>= greater than or equal

Numeric comparisons are made as follows:

- All negative numbers are "less than" zero or any positive number.
- Integers are converted to floating-point when a comparison is between mixed numeric data types. This conversion requires additional processing time.
- When an operator checks for equality or inequality of numeric expressions, use integers wherever possible. This is due to the inexactness, and rounding and truncation errors, of floating-point values.
- To check equality of floating-point numbers, compare the absolute value of their difference to a small enough limit. For example, use ABS(A − B) < 1E−15 instead of A = B.

RELATIONAL Operators

Examples

IF A = B THEN 1290

This example transfers program control to line 1290 if the contents of variable A equals the contents of variable B.

IF CS% <= BF% PRINT "Success!"

The message is printed if the contents of CS% are less than or equal to the contents of BF%. If BF% contains five, then the message is displayed if CS% contains five or anything less. If CS% is six or more, the PRINT portion of the instruction is skipped.

Statement

Usage

REM [remarks]
! [remarks]

Syntax Diagram

Description

The REM statement, or the ! character, allows remarks to be inserted into a program for documentation purposes.

- Either REM or ! may immediately follow the line number.
- Remarks may follow any program statement except DATA.
- When the REM form follows a program statement, it must be separated from the statement by the \ character.
- Anything following REM or the ! character is considered a remark, and is ignored by BASIC.

Example

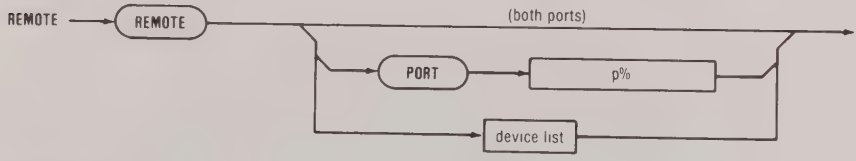
The following example illustrates some common uses of ! and the REM statement:

```
10 REM -- RESISTOR NETWORK VERIFICATION
20 !           R1276 - R1299
30 !
40 REM -- Program History
50 !           Rev. No.   Date       Author
60 !             1.0      1/21/79    B. Hansen
70 !             2.0      11/13/79   N. Kelly
499 REM -- end header
500 PRINT 'Mount Resistor On Fixture'          \REM Prepare for Test
510 PRINT 'Then Press RETURN';                  !Display prompt
520 INPUT A$                                   !Accept keyboard entry
```

Usage

REMOTE [PORT numeric expression]
REMOTE [device list]

Syntax Diagram



Description

REMOTE sets the REN (Remote Enable) line on the IEEE-488 bus to true.

- REN is set true on both ports if no port number is given.
- REN is set true only on the specified port when a port number is specified.
- REMOTE may be followed by a device list.
- When REMOTE is followed by a device list, REN is set true on the port or ports represented by instruments in the device list.
- After this, the listed devices are addressed to listen (sent an MLA message).
- Normally this places the instruments into remote mode, depending on the instrument design.

REMOTE
Statement

Example

In the following example, instrument port 0 is initialized at line 110. This sets REN true. However, the instruments on the bus are not in remote state until they have received a MLA message. Line 120 sends the addresses (2 and 4).

```
110     INIT PORT 0
120     REMOTE @ 2 @ 4
```

The following example is similar to the previous one. Note the use of the variable A% to designate the port number and B to designate the address of the device by adding A%*100 to B. This program correctly addresses the instrument without regard to which port it is on.

```
110     INIT PORT A%
120     REMOTE @ A% * 100 + B
```

The following statement sets REN true on both port 0 and 1.

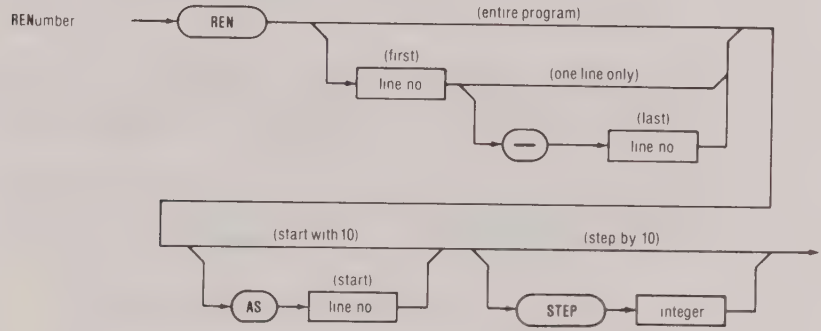
```
510     REMOTE
```

Immediate Mode Command

Usage

REN
REN [start-stop] AS [new start] STEP [step size]

Syntax



Description

The REN command changes the line numbers of some or all of the program lines in memory. Renumbering is useful to make room for additional program lines.

- REN cannot change the order of program lines.
- REN changes all references to line numbers (i.e., GOTO, GOSUB, etc.) in the program to reflect the new line numbers.
- All items shown in the syntax diagram are optional except the command word REN.
- The entire program is renumbered when no line numbers are specified.
- One line is renumbered when a single line number is specified. The command is ignored if the line does not exist.
- A portion of the program is renumbered when two line numbers are specified.

REN
Immediate Mode Command

- The line number following AS specifies the new starting number of the segment being renumbered. If this would rearrange the sequence of the program, a fatal error occurs and the line numbering remains unchanged.

- When AS is not specified, and a line number or range of line numbers is not specified following REN, the new starting line number is 10.

- When AS is not specified and a line number or range of line numbers is specified following REN, the new starting line number is the same as the old first line number of the range specified.

- The value of the integer expression following STEP must be positive. It defines the difference between any two consecutive, renumbered lines.

- If there is no STEP keyword, the line increment is 10.

- If the value of the integer expression following STEP is so large that the new line numbers would force the program to be rearranged, a fatal error occurs and the lines are not changed.

CAUTION

Renumbering from lower to higher line numbers (with more digits) may cause the renumbered lines to exceed the 79-character maximum line length allowed by BASIC. Restricting program lines to 74 characters maximum length will generally eliminate this problem. This exception is on long lines which include line number references (e.g., ON expression GOTO, IF-THEN-ELSE with line numbers, etc).

NOTE

Program lines containing the ERL (error line) function may have statements such as IF ERL = 200 THEN RESUME 400. The expression, the constant 200, is used as a line number reference. It is not changed during renumbering. It may need to be changed to the correct line number manually.

Examples

The following program is used in the renumbering examples below:

```
10   A = 1
20   PRINT A + A
30   A = A + 1
40   IF A <= 2 THEN 20
50   PRINT "Done!"
60   END
```

COMMAND	RESULT
`REN 60 AS 32767`	Change line 60 to read: `32767 END`

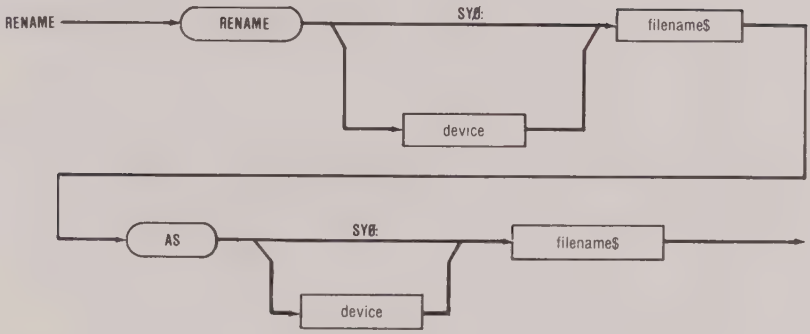
`REN 10-50 AS 5 STEP 5`	Change lines 10 through 50 to read:
	`5 A = 1`
	`10 PRINT A + A`
	`15 A = A + 1`
	`20 IF A <= 2 THEN 10`
	`25 PRINT "Done!"`
	Note the changed reference in line 20.

`REN`	This would in this case restore the program back to its original form.
`REN 60 AS 1000`	Renumber only line 60 as line 1000. An error results if any lines are numbered between 60 and 1001, since this would rearrange program sequence.
`REN 60 AS 1000 STEP 5`	Same as the previous example. STEP is ignored when only one line is renumbered.
`REN 10-500 AS 1000`	Renumber lines 10 through 500 to start at 1000, in steps of 10.

Usage

RENAME {oldfile$} AS {newfile$}

Syntax Diagram



Description

The RENAME statement gives a file a new name.

- Oldfile$ is the current filename and newfile$ is the new filename to be used.
- The device "SY0:" is assumed if none is given.
- The extension ".BAS" is assumed if an extension is omitted.

Example

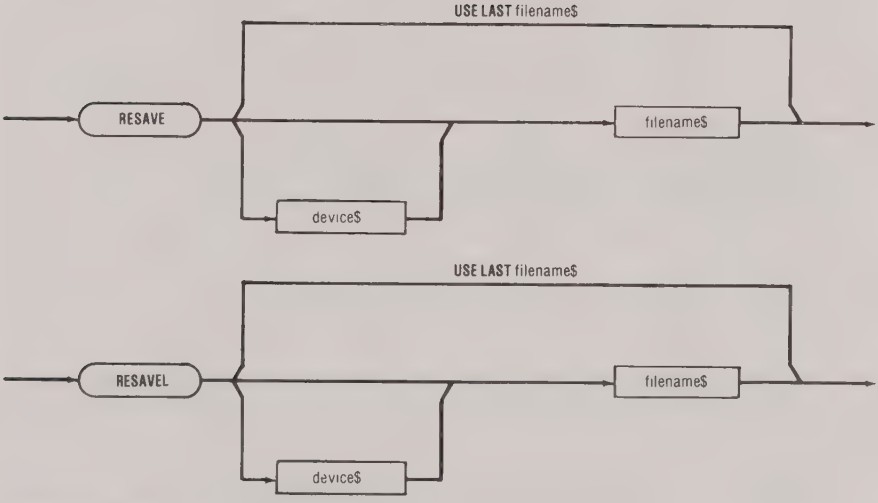
```
RENAME "PROG.FD2" AS "NPROG.FD2"
RENAME "MF1:TEST1" AS "MF1:TEST2"
```

RESAVE
RESAVEL
Statements

Usage

RESAVE [filename]
RESAVEL [filename]

Syntax Diagram



Description

The RESAVE and RESAVEL are an alternative to the SAVE and SAVEL commands. The SAVE and SAVEL commands ask whether or not the user wants to overwrite an existing .BAS or .BAL file. The RESAVE and RESAVEL commands will assume that any existing program file of the same name should be overwritten, and will not ask the user to confirm that the file should be clobbered.

- The RESAVE statement stores an ASCII program. See the SAVE statement.
- The RESAVEL statement stores an lexical format program. See the SAVEL statement.
- The RESAVE statement always uses the default file name extension .BAS. If your program uses a different extension than .BAS, you must specify the entire filename each time the RESAVE statement is used.

- The RESAVEL statement always uses the default file name extension .BAL.

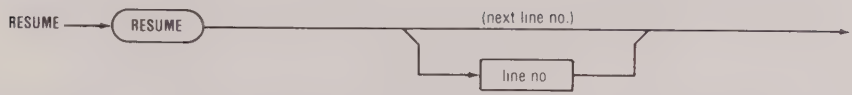
Example

```
Ready
OLD "TEST"

Ready
SAVE "TEST"
Replace existing file TEST.BAS? NO

Ready
RESAVE "TEST"

Ready
```

Usage

RESUME [line number]

Syntax Diagram



Description

The RESUME statement acknowledges an interrupt and allows program operation to resume with the next statement after the one being completed when the interrupt occurred or at another specified program location.

- RESUME (no line number) branches to the statement following the one being executed at the point where the interrupt occurred.
- If the interrupt occurred in a multiple-statement line, the program resumes with the next statement on the line. There are two exceptions:

  1. Recoverable errors: The program resumes at the beginning of the statement that caused the error.
  2. Input Warning errors 801, 802, and 803: The INPUT statement which caused the error requests the value to be entered again. It did not accept the erroneous entry.

- RESUME (line number) branches to the specified line number.
- RESUME terminates the interrupt handler routine.

Usage

RETURN

Description

The RETURN statement is used after a GOSUB statement to terminate the subroutine and return control to the program statement following the GOSUB.

- A RETURN statement without a matching GOSUB will cause error 701 (RETURN without GOSUB).

Example

```
10   PRINT "hello world"
20   GOSUB 100
30   PRINT "I am the greatest"
40   END
100  ! this is a subroutine
110  PRINT "To err is human, but it takes a computer ";
120  PRINT "to really blow it."
130  RETURN
```

Usage

RIGHT (string$, start%)

Description

The RIGHT function returns a substring of the specified string, (string$) starting from the specified character position, (string%) to the right end of the string.

- The RIGHT function returns a null string when the starting character position specified is greater than the string length.
- The RIGHT function returns a null string when the starting character position specified is longer than the string.
- RIGHT returns an identical string when the starting character position is specified as 0 or 1.

Example

In the following examples, the string A$ contains the characters "THIS IS THE FIRST EXAMPLE STRING".

STATEMENT	RESULT
`Y$ = RIGHT (A$, 27%)`	`Y$ = "STRING", the characters of A$ starting 27 characters from the left.`
`PRINT "RIGHT "; RIGHT (A$, 27%)`	`Displays "RIGHT STRING".`

Function

Format

RND

Description

The RND function produces a pseudo-random number that is greater than zero and less than one.

- The range of output values is positive floating-point values greater than zero and less than one.
- The sequence of values returned by RND is repeatable unless preceded by a RANDOMIZE statement.
- To expand the range of RND, multiply it by the desired range.
- To move the range of RND away from zero, add or subtract the desired movement.
- To include the endpoints (0 and 1), use:

  (INT (RND * (1E15 + 1))) / 1E15

Example

This example shows how to create a random number between 0 and 10.

```
3200 R=RND*10      ! Create a random number between 0 and 10.
```

The example below creates a random integer between 13 and 14.

```
4600 R%=INT(RND+13)
```

The program line below creates a random number in a compressed range between .5 and 1.

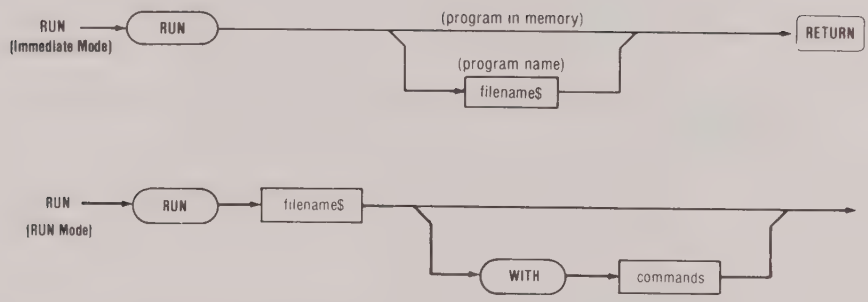
```
1115 R=(RND/2)+.5
```

Statement

Usage

RUN [filename]

Syntax Diagram



Description

The RUN statement restarts the program in main memory, or loads and runs a program available in file storage.

- Without a file name, RUN will restart the program in main memory.
- A file name may be specified as a quoted string or as a string expression.
- The WITH statement modifier replaces the FDOS command line with the string "filename$", followed by a space character, and the string "command$".
- If the WITH statement modifier is not used, the current FDOS command line remains unchanged.
- The program file will be searched for on the default System Device (SY0:) if the file name is not prefixed with a device name such as MF0:.
- Error 305 (file not found) results if the file is not found.
- When the file is located, it is loaded into main memory replacing the previous program, and control is transferred to it.

- Data stored in variables in the previous program is lost unless it was reserved in a common area with a COM statement or stored in a virtual array file.

- Virtual array files left open remain available to the next program, provided they are dimensioned in the new program identically and use the same variable names.

See the OLD command explain for loading ".BAL" in preference to ".BAS".

- If no filename extension is used, the BASIC interpreter first looks for a filename extension in ".BAL". If none is found, it then looks for a filename extension in ".BAS".

- A filename extension of ".BAL" requires less time to load into the Controller's memory than the same fit program with a ".BAS" extension. The lexial file requires less processing for conversion to internal form by the BASIC interpreter program.

Example

The following example searches for a program file named TEST2 on the System Device. If it is located, TEST2 is loaded into main memory and executed.

```
1050    RUN B$                  ! Chain to program named by B$
1060    END
```

The following example searches for a program file under a name stored in string B$. If the file is located, it is loaded into main memory and executed.

```
1050    RUN "TEST2"             ! Chain to program TEST 2
1060    END
```

The following example searches for a program file named "prog1" and replaces the FDOS command line with "PROG1" and "-a -b -c". The FDOS command line is then read and displayed.

```
10 !RUN WITH program example
20 RUN "prog1" WITH "-a -b -c"
30 end
```

This is the program that the previous program chains to:

```
10 !prog1.bas
20 !program chained to from previous example
30 PRINT CMDLINE$
40 end
```

Remarks

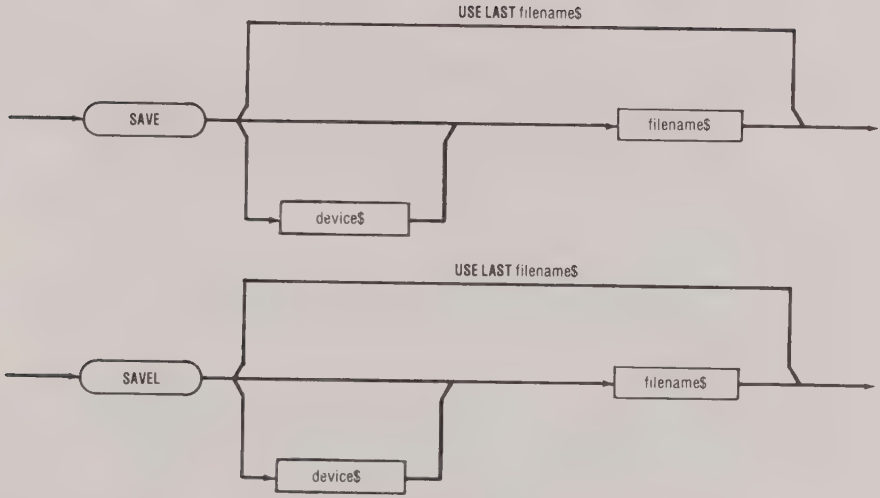
Compare the RUN statement with the OLD statement. Compare the RUN WITH statement with the EXEC WITH statement.

SAVE
SAVEL
Statements

Usage

SAVE [filename$]
SAVEL [filename$]

Syntax Diagram



Description

The SAVE and SAVEL statements are used to store the user program currently residing in memory as a file. SAVE stores the file in ASCII text form. SAVEL stores the file in lexically analyzed form. A discussion of these forms follows.

- A file name may be specified. It must be enclosed in single or double quotes. The file name may also be specified by a string variable.

- If no file name is specified, the same name will be used that was used in the OLD statement or the RUN statement that was used to load the program into memory.

- If no file name is specified for a new file, an error will occur.

- The file is stored on the default System Device if the file name is not preceded by MF0: for the floppy disk, or ED0: for the optional electronic disk. Refer to the Input and Output Statements section for a discussion of the default System Device concept.

- SAVE adds the file name extension .BAS when an extension is not specified.

- BASIC will request a confirming YES from the keyboard if an attempt is made to store a file under an existing file name. To avoid this interruption in a running program, use the RESAVE statement or first delete the file using a KILL statement.

- SAVE stores the ASCII form of the program in the largest available file space.

- SAVEL stores the lexical form of the program in the first available file space large enough to hold it. Note that this is different from SAVE.

- SAVE may also be used to obtain a printed listing of a program. To do so, specify the device name as either KB1: (RS-232-C Port 1) or KB2: (RS-232-C Port 2). The file name and extension are not required. See the User Manual for details on setting serial port baud rates.

In lexically analyzed form, a user program has binary numbers in place of ASCII character strings to represent line numbers, keywords, and operators. This form occupies less space and eliminates a processing step. Fluke BASIC programs in main memory are always in lexically analyzed form, even during editing. BASIC changes them to ASCII character form when needed for display or for storage by a SAVE statement.

Working copies of user programs should be saved in lexically analyzed form, using SAVEL. In this form, programs will occupy less file storage space and will load into memory quicker.

NOTE

A program saved via SAVEL may not be executable if the version of Fluke BASIC under which it was saved differs from the version under which it is to be executed.

The lexically analyzed form of a program cannot be displayed directly by the Utility program or sent to an external printer. Consequently, backup copies of user programs should be saved in ASCII character form, using SAVE. In this form, different versions of Fluke BASIC will be able to load and interpret the program.

Example

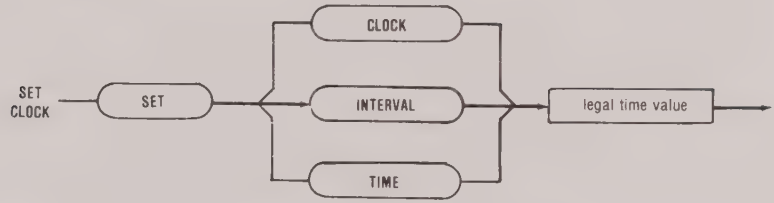
Examples of SAVE and SAVEL used in immediate mode follow:

STATEMENT	RESULT
`SAVE`	Save the program currently in memory, in ASCII character form, on the System device, using the same name that was used to OLD it into memory or RUN it. The default filename extension .BAS is always used.
`SAVE "TEST1"`	Save the program currently in memory, in ASCII character form, on the System Device, under the name TEST1.BAS.
`SAVEL 'MF0:TEST2'`	Save the program currently in memory, in lexically analyzed form, on the floppy disk (MF0:), under the name TEST2.BAL.
`SAVE 'ED0:DATA.T71'`	Save the program currently in memory, in ASCII character form, on the optional electronic disk (ED0:), under the name DATA.T71.
`SAVE "KB1:"`	Send the program currently in memory, in ASCII character form, to RS-232-C Serial Port 1.

Usage

SET CLOCK {time expression}

Syntax Diagram



Description

The SET CLOCK statement is used to indicate the time to be used for a timer interrupt. The SET CLOCK statement must be executed before an ON CLOCK statement, or an error 708 will occur.

The time may be expressed either in hours, minutes, and seconds, or as milliseconds (10 milliseconds minimum). The maximum time allowable is 24 hours. The following are all legal time values:

1:00	one hour
0:0:01	one second
0:10	ten minutes
hr:0:0	"hr" hours
tv	"tv" milliseconds
time+1000	current time plus 1 second

Note that the the "hh:mm:ss" form permits the seconds field to be omitted.

Clock interrupts are enabled by using a SET CLOCK statement to specify the time of day to be used, and activating the interrupt and selecting an interrupt handler by executing an ON CLOCK statement.

Example

The following sequence is used to activate the time of day interrupt:

```
100 SET CLOCK 10:00              ! set interrupt at 10 AM
110 ON CLOCK GOTO ...            ! activate interrupt
```

Usage:

SET CMDLINE$ {string expression}

Syntax Diagram:

Description

The SET CMDLINE$ statement sets (writes the string expression to) the FDOS command line.

- The maximum permissible string length in the command line is 80 characters. Error 328 is diagnosed (command line too long) if the length of the {string expression} is too long.
- If there are any imbedded carriage return characters [CHR$(13)] in the {string expression}, the command line set by FDOS is terminated at the point where the carriage return character occurs.

Example

The example program demonstrates the CMDLINE$ statement and truncation of the command line caused by an imbedded carriage return character.

```
10  SET CMDLINE$ "Hello world"
20  PRINT '"'; CMDLINE$ '"' \ PRINT
30  !
40  ! String with embedded carriage return. Watch this..
50  !
60  SET CMDLINE$ "Truncated" + CHR$(13%) + " here"
70  PRINT CMDLINE$
80  END
```

Running this program creates the following console display.

```
"Hello world"
Truncated
```

Remarks

See the entry for the CMDLINE$ function.

Usage

SET DATE ⟨date⟩

Syntax Diagram

Description

The SET DATE statement sets the contents of the system calendar.

- The SET DATE statement may be used from a BASIC program or from the immediate mode.
- The ⟨date⟩ parameter is expressed as

  ⟨day⟩ "," ⟨month⟩ "," ⟨year⟩

  in which the valid ranges are

Parameter	Range
⟨day⟩	1 .. [28..31] (depends on month)
⟨month⟩	1 .. 12
⟨year⟩	83 .. 99

- If an invalid date is specified a (recoverable) error 707 will be reported.

Example

Some examples of setting the date are:

```
SET DATE 11, 11, 83
```

to set the date on the 11th of November, 1983, and

```
SET DATE 3, 10, 84
```

to set the date on the 3rd of October, 1984.

Statement

Usage:

SET ECHO

Syntax Diagram:

Description

The SET ECHO statement is one of two statements used to control the keyboard mode. The SET ECHO statement puts the keyboard into normal "line" or "echo" mode, in which keystrokes are echoed by the Operating System, and in which the "line editing" keys (⟨CTRL⟩/U, ⟨CTRL⟩/R, DELETE, etc.) are active. In echo mode, a statement which reads a line (using the INPUT statement) will not be completed until a complete line has been entered by the user from the keyboard.

Example

```
1020 SET ECHO              ! Set echo keyboard mode
```

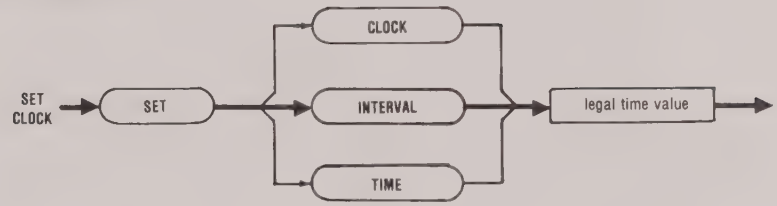
Remarks

Compare the SET NOECHO statement, which puts the keyboard into the editing ("character") mode in which no control characters (except ⟨CTRL⟩/S, ⟨CTRL⟩/Q, ⟨CTRL⟩/C, and ⟨CTRL⟩/P) are trapped by the Operating System.

Usage

SET INTERVAL time expression

Syntax Diagram



Description

The SET INTERVAL statement is used to indicate the interval between timer interrupts. The SET INTERVAL statement must be executed before the corresponding ON INTERVAL statement, or an error 708 will occur.

The interval time may be expressed either in hours, minutes, and seconds, or as milliseconds (10 milliseconds minimum). The maximum time allowable is 24 hours. The following are all legal interval values:

1:00	one hour
0:0:01	one second
0:10	ten minutes
hr:0:0	"hr" hours
tv	"tv" milliseconds

Note that the the "hh:mm:ss" form permits the seconds field to be omitted.

Interval interrupts are enabled by: Using a SET INTERVAL statement to specify the interval period to be used, and activating the interrupt and selecting an interrupt handler by executing an ON INTERVAL statement.

Example

A periodic interrupt is activated by:

```
120 SET INTERVAL 1500              ! set interrupt every 1500 ms
130 ON INTERVAL GOTO ...           ! activate interrupt
```

Usage

SET NOECHO

Syntax Diagram

Description

The SET NOECHO statement puts the keyboard into editing, or "character" mode in which no control characters (except ⟨CTRL⟩/S, ⟨CTRL⟩/Q, ⟨CTRL⟩/C, and ⟨CTRL⟩/P) are trapped by the Operating System. All characters except ⟨CTRL⟩/S and ⟨CTRL⟩/Q (which control the speed of data transfer to and from the keyboard) are returned to the BASIC program, but the ⟨CTRL⟩/C interrupts are still active in this state (they will also trap ⟨CTRL⟩/P). This mode prevents the Operating System from echoing characters, prevents certain control character recognition, and causes each individual key code to be returned as soon as the key is struck (rather than when a complete line has been entered). Character mode input is primarily useful for user-written editors which need both, to closely control the screen layout, and to handle single keystrokes (e.g., the "arrow" keys).

- In NOECHO mode, characters entered from the keyboard are not echoed to the display by the Operating System. The BASIC program itself must explicitly display (by using the PRINT statement) any characters which should appear on the screen.

- In NOECHO mode the ⟨CTRL⟩/U, DELETE, and ⟨CTRL⟩/R keys are not processed by the Operating System. Any processing of such characters must be done in the the BASIC program.

- Both ⟨CTRL⟩ C and ⟨CTRL⟩/P will cause a ⟨CTRL⟩ C interrupt to occur. This means that the Interpreter will return to Immediate Mode when either ⟨CTRL⟩/C or ⟨CTRL⟩ P is entered unless an ON ⟨CTRL⟩/C handler has been specified by the executing BASIC program. The ⟨CTRL⟩/C or ⟨CTRL⟩/P character codes may be returned to the program and may be processed using the INCHAR(0%) function.

SET NOECHO
Statement

- When the SET NOECHO statement is used, the BASIC program absolutely must not use an INPUT statement which does not use a channel number such as:

```
INPUT A$
```

  The use of this statement form will immediately put the keyboard back into the echo mode since the input statement operates on the basis of lines, not characters. (It is, of course, possible to use another SET NOECHO statement to put the keyboard back into noecho mode after the INPUT statement has been executed.) All noecho mode keyboard input must be performed using the INCHAR() function, which retrieves input one character at a time.

- Using the COPY statement when data is copied from device KB0: (for example, COPY "KB0:" TO "KB1:") will immediately place the keyboard in ECHO mode because the COPY statement reads lines from the keyboard.

- When the Interpreter returns to Immediate Mode the keyboard will be returned to echo mode; there is no way to defeat this process since the Interpreter reads lines from the keyboard.

Example

```
22055 SET NOECHO     ! Set the keyboard mode to "no echo"
```

Remarks

Compare the SET ECHO statement, which puts the keyboard into normal "line" or "echo" mode, in which keystrokes are echoed by Operating System, the "line editing" keys (CTRL U, CTRL R, DELETE, etc.) are active, and in which reads a line (using the INPUT statement) or a character (using the INCHAR() function) will not be completed until a complete line has been entered by the user from the keyboard.

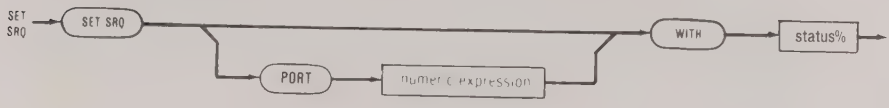
SET SRQ Statement

IEEE-488

Usage

SET SRQ [PORT {numerice expression}] {WITH status%}

Syntax Diagram



Description

The SET SRQ statement requests service from the current Controller in Charge (CIC) of the IEEE-488 bus interface by sending a Service Request (SRQ) message.

- status% is the "serial poll data byte" returned when the CIC performs a serial poll.
- The numeric expression for the IEEE-488 port must be either 0 (for the standard IEEE-488 port) or 1 (optional IEEE-488 port).
- If no port is specified, port 0 will be used.
- Both port numeric expression and status% must be numeric. They will be rounded to an integer if necessary. An error will be reported if:

  1. Either the port numeric expression or status% is not numeric, or
  2. An overflow occurs when either the port numeric expression or status% is converted to integer, or
  3. The value of p% is not in the closed interval [0..1], or
  4. The value of status% is not in the closed interval [0..255].

SET SRQ
Statement

Example

The statement

```
SET SRQ PORT 1 WITH 5
```

would send the SRQ message on port 1. When a serial poll is performed, the integer 5 [with an Request Service (RQS) bit] will be returned to the CIC (on port 1) as the status of the 1722A. (Note that the RQS bit is the 2^6 bit in the status byte.)

The statement

```
SET SRQ WITH ASCII("A")
```

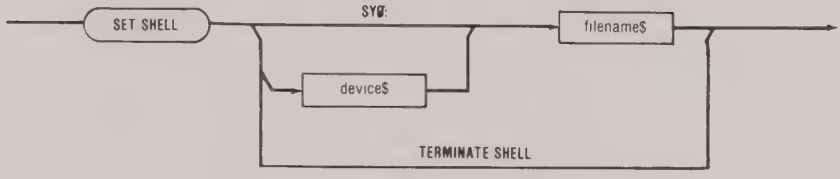
would send the SRQ message on port 0 with a status byte of 65 (corresponding to "A").

When the first serial poll following the use of the SET SRQ statement is performed, the RQS bit will be set; subsequent serial polls will cause the 1722A to respond with the same bit pattern less the RQS bit.

Usage

SET SHELL [device$][filename$]

Syntax Diagram



Description

The SET SHELL statement specifies a program other than the default Operating System user interface (FDOS prompt) to be used when a program exits.

- File$ is a string expression which specifies the filename of a program. If the string file$ is omitted, the default FDOS user interface will be used.

Example

The BASIC Interpreter can be used as a shell as follows:

```
SET SHELL "MF0:BASIC"
EXEC "MF0:FUP"
```

The statements will execute the File Utility program (FUP) and, when the FUP program terminates, return to the BASIC Intepreter.

Note that the effect of the SET SHELL statement can sometimes be confusing. Consider:

```
Ready
SET SHELL "MF0:BASIC"

Ready
EXIT

BASIC   Version 1.0

Ready
```

SET SHELL
Statement

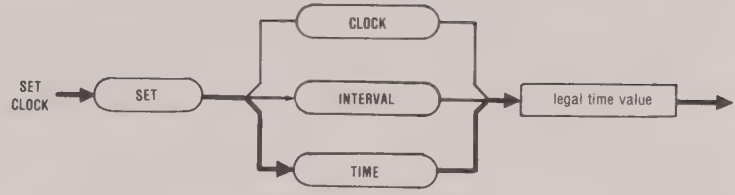
Note that when BASIC is the shell, an EXIT statement simply reloads the Interpreter. This can be corrected by using:

```
Ready
SET SHELL

Ready
EXIT

FDOS>
```

Usage

SET TIME [time expression]

Syntax Diagram



Description

The SET TIME Statement sets the value of the system clock.

- The time may be set from a BASIC program or from the immediate mode.
- The time may be expressed as:

  hours ":" minutes [":" seconds]

  or

  milliseconds

- The valid ranges for the parameters are:

Parameter	Range
hours	0 .. 24
minutes	0 .. 59
seconds	0 .. 59
milliseconds	0 .. 86400000

- Time is considered to use a 24-hour clock. An hours value of 24 is acceptable only if the minutes and seconds are both zero.
- If only a ⟨milliseconds⟩ term is specified, it is assumed to be milliseconds since midnight (with a 10-millisecond resolution).

SET TIME
Statement

Example

```
SET TIME 13:40:01          ! Set the clock to 1:40:01 PM.

SET TIME 40000             ! Set the clock to 40 seconds after
                           midnight

SET TIME TIME + 1000       ! Set the clock ahead 1000 millliseconds
```

Format

SGN (numeric expression)

Description

The SGN function returns the sign of a floating-point or integer number. This is called the signum function.

- SGN has a floating-point or integer number as an argument, and returns one of three integers: 1, 0, or –1.

  1 indicates a positive number.

  0 indicates a value of 0.

  –1 indicates a negative number.

- The domain of input values is any positive or negative floating-point or integer value, or zero.

Example

```
PRINT A, SGN(A)
27.093        1

Ready
PRINT B, SGN(B)
0             0

Ready
```

Format

SIN (angle in radians)

Description

The SIN function returns the sine of an angle that is expressed in radians.

- SIN has a floating-point number as an argument, and returns a floating-point result.
- The range of the input values is between and including the limits of ± 32767 radians. Error 607 results from input values outside these bounds.
- The range of output values is between and including the limits of –1 to +1.
- Input values within approximately 1E-16 of integer multiples of PI give a result of zero, rather than underflow error.

Example

The following example shows the SIN function being used to display the sine of an angle entered in degrees:

```
6605 ! Display the sine of an angle in DEGREES
6610 PRINT "Enter angle in degrees";          ! Print prompt
6620 INPUT A                                  ! Fetch the angle
6630 PRINT SIN(A*(PI/180))                    ! Print sine
```

SPACE$() Function

Format

SPACE$ (number of spaces)

Description

The SPACE$ function returns a string of spaces as specified.

- SPACE$ has an integer as an argument, and returns a string of spaces equal in length to its integer argument.
- SPACE$ is useful for formatting display and print outputs.

Example

The following example illustrates results of using SPACE$ to format a display.

```
10    N% = 5%
400   A$ = "This is the first example"
450   B$ = "STRING"
500   PRINT A$; SPACE$ (N%); B$
```

Display results:

```
This is the first example:  STRING
```

Format

SPL (device number)

Description

The SPL (Serial Poll) function performs a serial poll of a specified instrument and returns an integer status byte result. By sequentially performing serial polls of instruments and checking for SRQ in the status bytes, the SRQ routine can determine which instruments set SRQ. By examining the remaining bits of some instruments, the SRQ routine can determine why the SRQ bit was set and take appropriate action.

- SPL is the only way to reset the interrupt status of ON SRQ.
- The result will be from 0 to 255.
- SPL may be performed at any time, although it is normally used in an SRQ handling routine.
- When a serial poll is performed on an instrument, the instrument returns a status byte. Bit 6 of the status byte is set to 1 if the polled instrument is the one that requested service.
- The remaining bits may indicate other status information of the instrument. Consult the instrument manual for their meanings.
- The instrument asserting SRQ true should deassert SRQ when it is serial polled.
- More than one instrument can hold SRQ true at the same time.
- Refer to Appendix G, ASCII/IEEE-488 Bus Codes, for a chart of binary bytes and decimal numbers.
- The Request Service bit of the serial poll response may be tested by:

```
IF SPL(device) AND 64% THEN ...
```

SPL()
Function

Example

The following example performs a serial poll on device 16 with secondary address 13 (on port 0). Then the statement(s) following THEN are executed if a non-zero result is returned by the instrument.

```
5390    IF SPL(16%:13%) THEN ...
```

The following example assigns the result of a serial poll of device DV% on port P% to the variable Y%.

```
6820    Y% = SPL(P% * 100% + DV%)
```

As shown in Appendix G, 64 has the binary pattern 0100 0000 (bit 6 set). The following example uses AND 64% with the value of SPL to see which device caused the service request. The value of SPL is first saved in Y% and then manipulated several times in routine 5110 - 5190. This is necessary since the instrument asserting SRQ only gives its status once for this service request. Line 5130 checks for other status byte information.

```
5100    REM -- SRQ Handler - Port 0
5110    Y% = SPL(VM%)                   ! Voltmeter handling
5120    IF NOT (Y% AND 64%) THEN 5200
5130    Y1% = Y% AND (NOT 64%)          ! Get status apart from SRQ
5140    ! other statements to handle
5150    ! voltmeter status
5190    RESUME
5200    Y% = SPL(CN%)                   ! Counter handling
5220    IF NOT (Y% AND 64%) THEN 5300
5230    ! other statements to handle
5240    ! counter status
5290    RESUME
5300    ! statements for other instruments
```

SQR() Function

Format

SQR (numeric expression)

Description

The SQR (Square Root) function returns a floating-point number equal to the square root of the input value.

- SQR has a floating-point number as an argument, and returns a floating-point result.
- The square root is the value which, if multiplied by itself, produces the given input value.
- The domain of input values is positive numbers and zero. Error 604 results when a negative number is used as input.

Example

```
PRINT SQR(49)
7

Ready
```

Immediate Mode Command

Format

STEP

Description

The Immediate Mode STEP command sets a mode in which each statement within a program is executed individually by pressing RETURN.

- STEP must first be enabled by a breakpoint stop in a running program, caused by STOP ON or CONT TO.
- After a breakpoint stop, type STEP to select Step Mode.
- From Step Mode, type CTRL C or any Immediate Mode command to return to Immediate Mode.
- Any BASIC command or statement that is available in Immediate Mode can also be used to exit Step Mode.
- In Step Mode, one statement is executed each time RETURN is pressed.
- After executing each statement, the display reads: STOP ON LINE n, where n is the next line to be executed.
- When used with a variable TRACE ON, the display will also show changes in selected variables whenever a statement assigns a new variable value.

Format

STIME$

Description

The function STIME$ (seconds time) returns the current time of day as the eight-character string

hh:mm:ss

in which

hh is the hour (24-hour format)

mm is the minute within the hour

ss is the second within the minute.

Any fractional seconds component is truncated.

Example

This example prints the time (including seconds) in the Immediate Mode.

```
PRINT STIME$
10:03:48
```

If the time is really 11:29:53.6 (i.e., 53.6 seconds past the minute) the string returned by STIME$() is 11:29:53.

Comparisons

Description

Strings are compared on the basis of character matching, order within the standard ASCII code set, and overall length.

- A string character value is its relative position in the ASCII code table (see Appendix I). This means, for example, that all capital letters are less than any lower-case letter.
- String comparisons are done left to right, character for character. The first inequality determines "less than" or "greater than".
- If no inequality is found, the shorter string is "less than".
- Strings must be identical to be equal.
- The operators are the same as the numerical operators, with the addition of a concatenation operator.

  = equal
  < less than
  > greater than
  <> not equal
  ≤ less than or equal
  ≥ greater than or equal

  \+ concatenates (connects) strings together

STRING
Comparisons

Examples

The following examples illustrate the rules for string comparisons:

TRUE COMPARISON	REASON
`"" < any other string`	"" is a string with no elements.
`" Z" < "AZ"`	Space is less than "A".
`" Z" < "A"`	Space is less than "A".
`"LONG STRING" < "LONGER STRING"`	Space is less than "E".
`"1999" < "20"`	"1" is less than "2".
`"ON" < "ONE"`	"ON" has fewer characters.

These are examples using the concatenation operator:

EXPRESSION(S)	RESULT
`"BEGIN" + "OPERATION"`	`BEGINOPERATION`
`"BEGIN " + "OPERATION"`	`BEGIN OPERATION`
`A$ = " Volts"` `"ENTER" + A$`	`ENTER Volts`
`A$ = "Millivolts"` `"ENTER" + A$`	`ENTERMillivolts`

Description

Strings are sequences of 8-bit positive integers that are normally interpreted as ASCII characters. Strings are used to store characters for messages to instruments and to the display, as well as for storage of binary data taken from instruments. String data has the following characteristics:

- Maximum length limited only by available memory or 16383 characters, whichever is smaller.
- Memory requirement: each string of 16 or less characters occupies an 18-byte memory segment and an additional 18 bytes for each additional 16 characters.
- String data is normally displayed by the Instrument Controller in ASCII. See the Touch-Sensitive Display section for exceptions.
- When interpreted as ASCII, the value of the most significant (8th) bit is ignored.

String Constants

String constants are expressed as a sequence of printable characters (numerics, uppercase alphabetic characters, lowercase alphabetic characters, printable symbols, e.g., *, –, [, etc). In most cases, string constants must be enclosed in either single or double quotes. Enclosing the statement in single quotes allows the use of double quotes in the constant and vice versa. String constants need not be expressed in quotes when part of a DATA statement or when entered after an INPUT statement. Some examples of strings are:

```
A$ = "The result of 3.8 * PI is "
IN$ = 'Reply with "YES" or "NO" '
```

String Variables

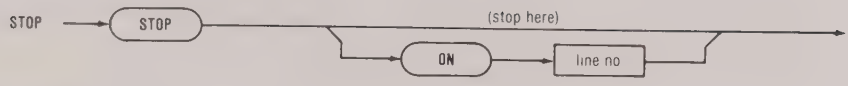
String variables are designated by a floating-point variable name followed by a "$" character.

Statement

Usage

STOP

Syntax Diagram



Description

The STOP statement halts execution of the BASIC program and displays the line number where the STOP occurred.

- STOP terminates program execution.
- STOP can be used to indicate "dead end" code branches, either because of errors or because of logical structuring.

Example

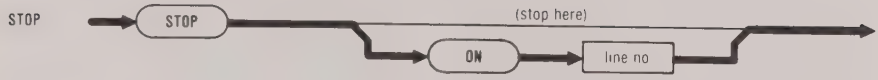
The following example illustrates a common use of STOP:

```
10 REM -- TEST PROGRAM
20 ! Other statements
30 !
999 STOP                              ! End of main procedure
1000 REM -- SUBROUTINES
1010 ! Other statements
5000 END
```

Usage

STOP ON {line number}

Syntax Diagram



Description

The STOP ON statement, stops execution of a program.

- STOP ON line number, allows a program to be run in sections during logic debugging.
- The program stops at the line number of the STOP when ON line number is not included.
- The program stops at the line number following ON, without executing it, when ON line number is included.
- STOP ON may be executed in either Immediate or Run Mode.
- STOP ON line number enables the STEP command.

Description

System variables store changing event information for use as required by a program. They are accessed by name and return a result in floating-point, integer, or string form as appropriate. The table below lists the system variables and gives their meaning and form.

System Variables

NAME	TYPE	EXAMPLE	MEANING
DATE$	String	08-Feb-81	Current date in the format DD-MM-YY.
ERL	Integer	1120	Line number at which the most recent BASIC program error occurred.
ERR	Integer	305	Error code of the most recent error the BASIC interpreter found in the program being executed.
FLEN	Integer	6	Length of the last file opened in 512-byte blocks.
KEY	Integer	20	Position number of the last Touch-Sensitive Display region pressed.
MEM	Floating-Point	29302	Amount of unused main memory, expressed in bytes.
RND	Floating-Point	0.2874767	Pseudo-random number greater than 0 and less than 1. Repeatable if not preceded by RANDOMIZE.
TIME	Floating-Point	0.5491405E+08	Number of milliseconds since the previous midnight.

SYSTEM
Variables

TIME$	String	17:45	Current time of day in 24-hour format.
STIME$	String	17:45:19	Current time of day, including seconds.
CMDFILE	Integer	–1	Command file active.
CMDLINE$	String	BASIC	Current Operating System command line.

Format

TAB (column number)

Description

The TAB function returns a string of spaces that would advance the current print position on an external printer to one column past the specified column number.

- TAB has an integer as an argument, and yields a string of spaces that may be preceded by Carriage Return and Line Feed.
- Any positive integer may be used (0 to 32767). TAB does not limit the length of a line.
- Because TAB is intended for an external printer, column position may be different than the display cursor.
- The current print position used by TAB to compute the number of spaces required is the total number of characters since the last Carriage Return and Line Feed.
- The display cursor column position will be different from the current print position when the current line contains a display control command, such as CPOS.
- The count for current print position includes every character transmitted, and does not assume that the printer responds to any positioning commands.
- A Carriage Return and Line Feed sequence precedes the string of spaces when the current print position is more than one column beyond the specified column number.
- TAB returns a null string when the current print position is one greater that the specified column number.

TAB()
Function

Example

The following examples illustrate the results of uses of the TAB function with the display. The display cursor position is initially at column 1.

```
PRINT TAB(5), "HELLO"
```

Advance 5 spaces (column 6) and display "HELLO".

```
PRINT "HELLO"; TAB(3); "THERE"
```

Display "HELLO", move the cursor down one line, and display "THERE" without leading spaces. Column 4 (TAB (3) + 1) is left of the cursor after displaying "HELLO".

```
PRINT CPOS(6, 3); TAB(5); "HELLO"
```

Display "HELLO" at the start of line 7. Column 6 (TAB (5) +1) is left of the current print position since CPOS (6,3) is an 8-character string, bringing the current print position to 9, even though the display cursor moves to row 6, column 3. All CPOS statements result in an 8-character string. That is: LEN (CPOS(r,c))=8.

```
PRINT CPOS(6, 3); TAB(13); "HELLO"
```

Display "HELLO" at column 8 on line 6. Since CPOS (6,3) is an 8-character string, the current position is 9 when the TAB to column 14 (13 + 1) is calculated. TAB returns 5 spaces (14 - 9), moving the display cursor from its actual column 3 position to column 8. All CPOS statements result in an 8-character string. That is: LEN (CPOS(r,c))=8.

The following example displays the numbers 100 through 154 in 10 columns that are each 8 spaces wide. The column number index is added to the bottom for reference. The extra space to the left of each column is for the sign (blank = positive). The PRINT statement in line 50 advances the cursor to the next line. Refer to the section General Purpose Fluke Basic Statements for a discussion of FOR-NEXT program loops.

```
10      FOR I = 100 TO 150 STEP 5
20      FOR K = 0 TO 4
30      PRINT TAB(8 * K); I+K;
40      NEXT K
50      PRINT
60      NEXT I
```

Results:

```
 100  101  102  103  104
 105  106  107  108  109
 110  111  112  113  114
 115  116  117  118  119
 120  121  122  123  124
 125  126  127  128  129
 130  131  132  133  134
 135  136  137  138  139
 140  141  142  143  144
 145  146  147  148  149
 150  151  152  153  154
```

TAN() Function

Format

TAN (angle in radians)

Description

The TAN function returns the tangent of an angle that is expressed in radians.

- TAN has a floating-point number as an argument, and returns a floating-point number.
- The range of input values is between and including the limits of ± 32767 radians. Error 607 results from input values outside these bounds.
- Input values within approximately 1E-16 of integer multiples of PI/2 result in error 603. The tangent function has infinite discontinuities at these points.

Example

The following example displays the tangent of an angle that is entered in degrees:

```
6900 ! Display tangent of angle (degree units)
6910 PRINT "Enter angle in degrees";          ! Print prompt
6920 INPUT A                                  ! Fetch angle
6930 PRINT TAN(A(PI/180))                     ! Display result
```

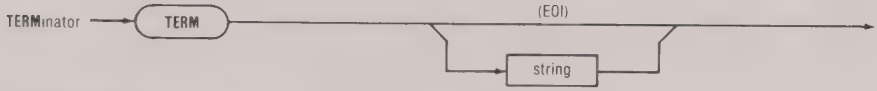
Statement

IEEE-488

Usage

TERM [terminating character]

Syntax Diagram



Description

TERM (TERMinator) may be used to specify an 8-bit character, which when received from an IEEE-488 bus device in response to an INPUT statement, will act as a line terminator for the input data.

- The EOI (End Or Identify) line on the Bus will always terminate input regardless of the use of the TERM statement.
- TERM allows the user to specify an arbitrary 8-bit byte that will also terminate input.
- The terminating character is Line feed when TERM is not used.
- The terminating event is limited to the EOI bus control line when TERM is used without a character specified. In this mode, all 8-bit values are acceptable as input.
- Only one terminating character may be specified at a time.

Example

The following examples illustrate common uses of TERM:

STATEMENT	MEANING
`TERM '?'`	Select the question mark character as an input terminator.
`TERM CHR$(255%)`	Select a character with all bits set to one (in an 8-bit byte) as the input terminator.
`TERM`	Limit input termination to the EOI line of the Bus.
`TERM ''`	Null character specification. Limit input termination to the EOI line of the Bus.

TIMEOUT 154
Statement

IEEE-488

Usage:

TIMEOUT [numeric expression]

Syntax Diagram:

Description

TIMEOUT sets a limit on the amount of time the Instrument Controller will wait for a response to an IEEE-488 bus request. This prevents an instrument fault from halting the system.

- Error 408 results when the specified time is exceeded.
- The time allowed is 20 seconds when TIMEOUT is not used.
- The expression designates wait time in milliseconds.
- The range is 0 to 32767 milliseconds. Actual resolution is 10 milliseconds.
- Timeout protection is disabled by specifying zero milliseconds. This is especially useful during analysis of commands on the bus with a bus analyzer.

NOTE

The length of timeout at the IEEE-488 port can also be changed by the SET Utility program. Any time up to 255 seconds (4 minutes, 15 seconds) can be specified.

The following examples illustrate common uses of TIMEOUT.

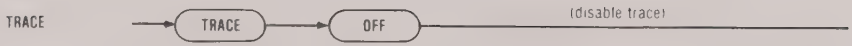
STATEMENT	RESULT
`TIMEOUT 500`	Set timeout limit to 500 milliseconds.
`TIMEOUT 0%`	Disable timeout protection

TRACE OFF 154A
Statement

Usage

TRACE OFF

Syntax Diagram



Description

TRACE OFF disables any pending or active trace assigned in the program and destroys the variable list:

Examples

The following example illustrates that TRACE ON starts only a line number trace after TRACE OFF:

```
10      TRACE ON A, B                   ! Trace variables A and B
50      TRACE OFF                       ! Halt the variable trace
100     TRACE ON                        ! Start a line number trace
```

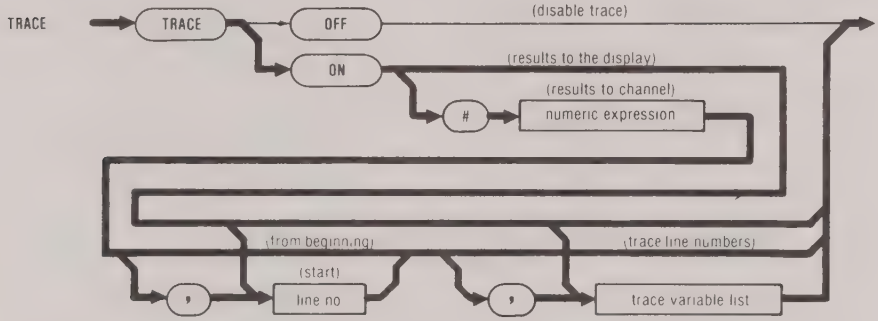
The following example illustrates a way to suspend tracing until a later point in a program.

```
10      TRACE ON A, B                   ! Trace variables A and B
50      TRACE ON 100                    ! Stop trace until line 100
100     ! Resume tracing variables A and B
```

Usage

TRACE ON [linenumber] [trace variable list]
TRACE ON [channel #n] [linenumber] [trace variable list]

Syntax Diagram



Description

TRACE prints a record of line numbers encountered or changes in variable values.

- If a previously opened channel is specified, the results of the trace are sent to the channel. Otherwise, the results are sent to the display.
- A starting line number for tracing may be specified. If it is not specified, tracing starts with the first line following the execution of the TRACE statement.
- Tracing is activated when the specified start line or the first line is encountered.
- TRACE may be used in either Immediate or Run Mode.

TRACE ON
Statement

Line Number Tracing

A line number trace has the following forms:

STATEMENT	MEANING
`TRACE ON`	Trace line numbers from the first line and send the results to the display.
`TRACE ON line number`	Trace line numbers from the specified line and send the results to the display.
`TRACE ON # channel`	Trace line numbers from the first line and send the results to the open channel.
`TRACE ON # channel, line number`	Trace line numbers from the specified line and send the results to the display.

- A line number trace and a variable trace will not execute concurrently.
- A line number trace occurring after a variable trace specifies a new line number after which variable tracing will resume, provided no TRACE OFF occurred in the interim.

Example

The following examples illustrate the results of different forms of line number trace statements.

STATEMENT	RESULT
`30 TRACE ON`	Start a line number trace at the next line following line 30.
`500 TRACE ON 1275`	Start a line number trace when line 1275 is reached.
`750 TRACE ON # 3%, 400`	Start a line number trace when line 400 is reached. Send the trace output to channel 3.

The line number trace displays a series of numbers representing the line numbers or the statements executed. The following example illustrates typical results.

Program

```
10      TRACE ON
20      I% = 0%
30      I% = I% + 1%
40      IF I% < 3% THEN 30
50      TRACE OFF
60      END
```

Results

```
 20
 30
 40
 30    Showing that the loop was
 40    executed 3 times
 30
 40
 50

Ready
```

TRACE ON
Statement

Variable Tracing

A variable trace has the following forms:

STATEMENT	MEANING
`TRACE ON variable list.`	Trace changes in value of selected variables from the first line, and send the results to the display.
`TRACE ON #channel, variable list`	Trace changes in value of selected variables from the first line, and send the results to the open channel.
`TRACE ON line number, variable list`	Trace changes in value of selected variables where the specified line is encountered, and send the results to the display.
`TRACE ON #channel, line number, variable list`	Trace changes in value of selected variables from the specified line, and send the results to the open channel.

- If a list of variables is specified, the trace is of changes in values of those variables. Otherwise, the trace is of line numbers encountered.
- A variable trace may specify one or more variables of any type: string, integer, and floating point.
- A variable trace of an array may use the form A() as the variable. A() means "TRACE ON all elements of array A".

- An array must be previously dimensioned before tracing.
- A variable trace and a line number trace will not execute simultaneously.
- Two or more variable traces will execute simultaneously. For example, TRACE ON A followed by TRACE ON B is equivalent to TRACE ON A,B.
- A variable trace occurring after a line number trace turns off the line number trace.

A variable trace statement resembles the line number trace statement except that a list of variable names is included. The following example specifies trace output to channel 2, tracing to start at line 340, and tracing of changes in values of A%, element (3,4) of array B, and all of array A$:

```
30      TRACE ON #2%, 340, A%, B(3%, 4%), A$()
```

NOTE

A variable trace of an array cannot be done without first dimensioning the array with a DIM statement.

Variable trace display output takes the following form:

line number identifier type(indices) = new value

Where:

1. Line number is the number of the line in which the variable was assigned a new value.
2. Identifier is the name of the variable.
3. Type is % for integers, $ for strings.
4. Indices identify which element of the array is being traced on and displayed (for array elements only).
5. New value is the new value assigned.

TRACE ON
Statement

This example shows the result of a trace of an array variable:

```
TRACE ON A (1, 2)                    Displays the value of
                                     A(1,2) when it is assigned.
                                     For example,

                                     220 A(1,2) = 47.3386
```

This example program illustrates the display resulting from a trace of an integer array program:

```
10      DIM A% (2%, 2%)
20      TRACE ON A%()
30      FOR I% = 0% TO 2%
40      FOR J% = 0% TO 2%
50      A% (I%, J%) = I% * J%
60      NEXT J%
70      NEXT I%
80      TRACE OFF
90      END
```

Results

```
50  A%(0,0) = 0
50  A%(0,1) = 0
50  A%(0,2) = 0
50  A%(1,0) = 0
50  A%(1,1) = 1
50  A%(1,2) = 2
50  A%(2,0) = 0
50  A%(2,1) = 2
50  A%(2,2) = 4
```

Other Trace Options

TRACE ON line number can be used to define a trace region within a program. The example below traces the array A$ only in the subroutine starting at line 110. Until TRACE OFF is executed, TRACE ON continues to trace all variables for which a TRACE ON was executed, and continues to send trace output to the specified channel or the display.

TRACE ON
Statement

```
10      DIM A$ (5%, 5%)
20      TRACE ON 110, A$()              ! Start tracing array A$ at
                                        line 110
30      !
40      FOR I% = 0% to 5%
50      A$ (I%, 0%) = CHR$ (ASCII (' ') + I%)
60      GOSUB 110
70      NEXT I%
80      TRACE OFF
90      STOP
100     !
110     FOR J% = 1% TO 5%
120     A$ (I%, J%) = A$ (I%, J% - 1%) + CHR$ (ASCII(' ') + I%
        + J%)
130     NEXT J%
140     TRACE ON 110
150     RETURN
160     END
```

The following trace display output results from running this program. Refer to Appendix I, ASCII/IEEE-1978 Bus Codes, and note the display characters that follow SPACE, character number 32, for clarification of these results.

```
120     A$(0,1) = !
120     A$(0,2) = !"
120     A$(0,3) = !"#
120     A$(0,4) = !"#$
120     A$(0,5) = !"#$%
120     A$(1,1) = "
120     A$(1,2) = "#
120     A$(1,3) = "#$
120     A$(1,4) = "#$%
120     A$(1,5) = "#$%&
120     A$(2,1) = #
120     A$(2,2) = #$
120     A$(2,3) = #$%
120     A$(2,4) = #$%&
120     A$(2,5) = #$%&'
120     A$(3,1) = $
120     A$(3,2) = $%
120     A$(3,3) = $%&
120     A$(3,4) = $%&'
120     A$(3,5) = $%&'(
120     A$(4,1) = %
120     A$(4,2) = %&
120     A$(4,3) = %&'
120     A$(4,4) = %&'(
120     A$(4,5) = %&'()
120     A$(5,1) = &
120     A$(5,2) = &'
120     A$(5,3) = &'(
120     A$(5,4) = &'()
120     A$(5,5) = &'()*
```

It is also possible to send trace output to different channels. Output is sent to one channel at a time. The following example illustrates this:

```
10      OPEN "TRACE1.DAT" AS NEW FILE 1%! First trace channel
20      OPEN "TRACE2.DAT" AS NEW FILE 2%! Second trace channel
100     TRACE ON A%, B$(), C()          ! Send output to console
250     TRACE ON #1%                    ! Send output to channel 1
370     TRACE ON #2%                    ! Send output to channel 2
1010    TRACE OFF                       ! Discontinue all tracing
1020    END
```

TRIGONOMETRIC Functions

156

Description

Fluke BASIC includes four trigonometric functions: sine, cosine, tangent, and arctangent. These functions have some discontinuities and limits that can produce unexpected errors when used improperly. The discussion that follows defines these factors.

Radian Angular Measure

Trigonometric functions in Fluke BASIC use radian measure for angular quantities. This is a simple concept that fits trigonometric calculations better than the use of degrees.

- The value PI stored by Fluke BASIC (3.14159265358979) represents an approximation of the ratio between the circumference and the diameter of any circle.
- Radian measure defines the angular distance around a circle as 2*PI radians. This is the equivalent of 360 degrees.
- Angles are thus easily defined as fractional parts of PI. For example: PI radians = 180 degrees PI / 2 radians = 90 degrees
- To convert from degrees to radians, multiply by (PI / 180).
- To convert from radians to degrees, multiply by (180 / PI).

Functions Available

The available trigonometric functions are:

SIN Function
COS Function
TAN Function
ATN Function

See the individual descriptions of these functions for details.

Statement

IEEE-488

Usage

TRIG {device list}

Syntax Diagram

Description

The TRIG (TRIGger) statement addresses a set of instruments as listeners and then triggers them simultaneously. The effect of the trigger is dependent upon the instrument. For example, a digital multimeter may take a reading, or a source instrument may go from standby to operate.

- TRIG addresses the specified devices as listeners.
- The controller then sends a GET (Group Execute Trigger) message on the bus or buses represented in the device list.

Examples

The following examples illustrate common uses of TRIG:

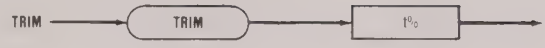
STATEMENT	RESULT
`TRIG @ 22% @ 3%`	Trigger devices 22 and 3 on port 0.
`TRIG @ 1% @ 105%`	Trigger device 1 on port 0, and device 5 on port 1.

Statement

Usage:

TRIM integer

Syntax Diagram:



Description

The TRIM statement specifies how trailing null (zero) bytes will be handled when using string variables from virtual-array files.

- The integer value determines how null bytes will be trimmed as follows:

  0 A zero value specifies that no trimming will be done.

  non-zero Any non-zero value specifies that all trailing zero bytes will be removed, which makes the virtual array string variables appear to be just like normal string variables.

- A floating-point value will be rounded to an integer.

Example

```
TRIM 1          ! Activate trailing null byte suppression
TRIM 0          ! Stop trimming null bytes
```

UCASE$ Function

Usage

UCASE$=(argument string$)

Description

The UCASE$ function converts strings to upper case (capitals).

- The argument must be a string variable.
- The output is a string (of the same length as the argument string$) with all alphabetic characters converted to upper-case.

Example

This example shows a string which is received from the keyboard being converted to upper case before decoding.

```
120 INPUT A$                        ! Fetch a string from the user
130 AU$=UCASE(A$)                   ! and convert it to upper case
131                                 ! before use.
140 IF ASCII$(AL$)>64 AND ASCII$(AL$)<69 THEN 1000 ELSE 120
141                                 ! Look for A-D only
```

Usage

UNLINK {filename$}

Syntax Diagram

Description

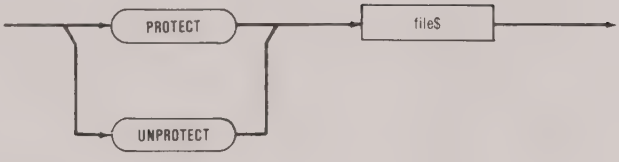
The UNLINK statement removes all reference to Assembly Language or FORTRAN subroutines from memory, making the previously reserved memory space available for other uses. Individual Assembly Language or FORTRAN subroutines cannot be selectively removed from memory.

- Memory freed by UNLINK is available for other uses.
- UNLINK can be used in either the Immediate or the Run mode.

Usage

UNPROTECT {file$}

Syntax Diagram



Description

The UNPROTECT statement changes the protection code for a specified file to allow it to be deleted (e.g., via KILL or OPEN AS NEW FILE).

- The "–" protection code is visible using the DIR or EDIR statements to examine the directory containing the specified file.
- Attempting to delete a protected file without using the UNPROTECT statement will result in an error.

Example

The instruction

```
10125 UNPROTECT "CSFILE.TMP"
```

assigns a "–" protection code to the file RPFILE.TMP to allow it to be deleted.

Format

VAL (string)

Description

The VAL function returns the floating-point numeric value of a numeric string.

- VAL has a string as an argument and yields a floating-point number.
- The string argument to VAL should be a legal floating-point number.
- String input consisting only of spaces, or a null (empty) string, returns the value 0.
- Spaces and Nulls may precede the input numeric string.
- Spaces, Nulls, Carriage Returns, and Line Feeds may follow the input numeric string.
- Error 803 results when the input string contains non-numeric elements.

Example

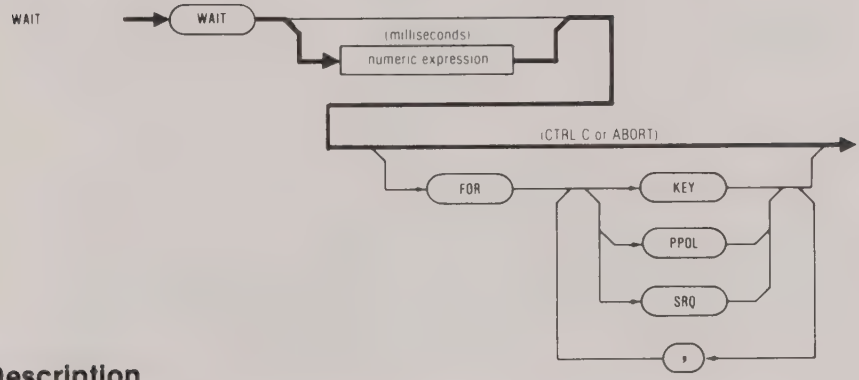
The following examples illustrate results of common uses of VAL.

STATEMENT	RESULTS
`Y = VAL (A$)`	The value of the numeric string A$ is assigned to the floating-point variable Y. For example, if A$ contains "1000", Y is assigned the value 1000.
`Z = VAL (" ")`	Assign Z the value 0. (The string consists only of spaces).

Usage

WAIT [time expression]

Syntax Diagram



Description

WAIT suspends program execution until either the specified time period elapses. Other forms of the WAIT statement allow waiting for an event (KEY, PPOL, or SRQ) to occur. These forms of the WAIT statement are described separately and as a group elsewhere in the Reference Section of this manual.

- Wait time is indefinite if a time is not specified.
- The Wait time (expressed in milliseconds) should be a positive integer, an expression that evaluates to a positive integer, or a legal time value. The WAIT statement is ignored if the wait time is zero or negative.

WAIT
Statement

- The time may be expressed either in hours, minutes, and seconds, or as milliseconds (10 milliseconds minimum). The maximum time allowable is 24 hours. The following are all legal time expressions.

expression	meaning
1:00	one hour
0:0:01	one second
0:10	ten minutes
hr:0:0	"hr" hours
t	"t" milliseconds
time+1000	current time plus 1 second

Note
The "hh:mm:ss" form permits the seconds field to be omitted.

- The minimum wait time is 10 milliseconds.
- Clock resolution is 10 milliseconds.
- Interrupt processing that has been previously activated by ON SRQ, ON PPOL, or ON KEY remains active, capable of terminating the wait.
- When an interrupt that has been enabled by an ON statement occurs, the program branches to the interrupt handling routine. The RESUME statement at the end of that routine branches back to the statement following WAIT.
- ⟨CTRL⟩ C or the ABORT switch will terminate the WAIT. If the program does not include a ⟨CTRL⟩ C handler, the program will be halted and control returned to Immediate Mode.

Examples

The following examples illustrate the uses of the WAIT statement.

```
4490 WAIT 500          !wait 500 milliseconds
5000 WAIT 13:00        !wait 13 hours
6000 WAIT 00:00:10     !wait 10 seconds
```

The following example illustrates the use of WAIT in requiring program execution to halt for the period of time, in milliseconds, supplied by the integer variable D%.

```
580      WAIT D%
```

Often a program must take into account external events, such as the length of time a programmable instrument needs to respond to program data. Use the WAIT statement to ensure proper timing of the test system. This example sends program data to a Fluke Model 6070B frequency synthesizer, then waits 1 second for the instrument to settle before going to the next program step.

```
50 Print @ 1 "FR200MZ, AP!V"   ! 200 MHz, 1 Volt amplitude
60 WAIT 1000                   ! wait 1 second
.
.
.
1000 RESUME
```

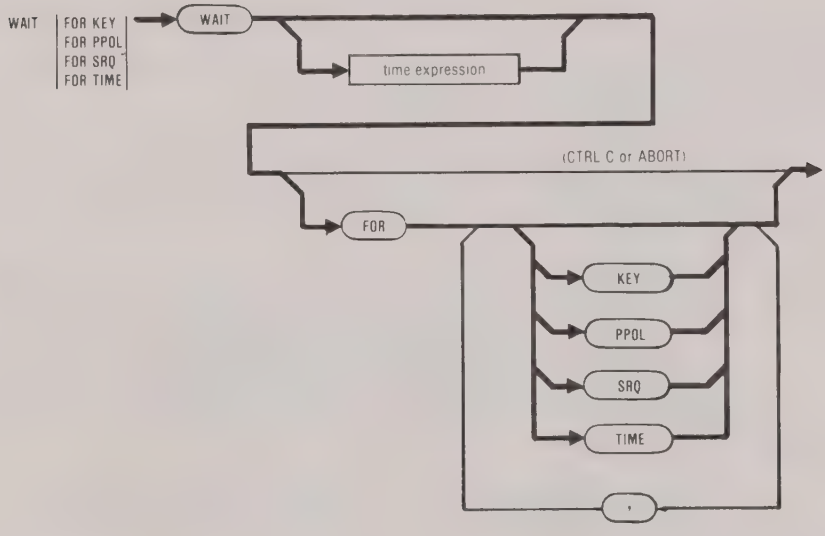
Statement

IEEE-488

Usage

WAIT [time expression] FOR | KEY
PPOL
SRQ
TIME |

Syntax Diagram



Description

The WAIT [time expression] [FOR event] statement suspends program execution until the specified interrupt event occurs or the specified time elapses.

- The interrupt is implicitly acknowledged by its occurrence.
- WAIT may be followed by a time expression specifying a period of time to wait for the interrupt.
- The time may be expressed either in hours, minutes, and seconds, or as milliseconds (10 milliseconds minimum). The maximum time allowable is 24 hours. The following are all legal time expressions.

WAIT
Statement

IEEE-488

expression	meaning
1:00	one hour
0:0:01	one second
0:10	ten minutes
hr:0:0	"hr" hours
t	"t" milliseconds
time+1000	current time plus 1 second

Note

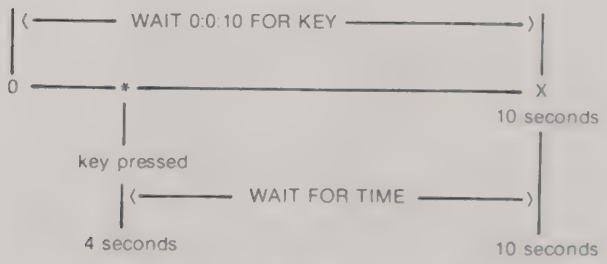
The "hh:mm:ss" form permits the seconds field to be omitted.

- When the specified time elapses, interrupt checking stops and the program continues with the next statement.

- An interrupt previously enabled by an ON-GOTO statement remains enabled during the waiting period, whether or not the WAIT statement references it.

- When a [time expression] is used with the WAIT statement and the specified interrupt occurs before the expiration of the [time expression], program execution can be delayed by the remainder of the time denoted by the [time expression] by using the WAIT FOR TIME statement. The following diagram and program fragment displays the relationship between the WAIT and WAIT FOR TIME statements.

```
500 WAIT 0:0:10 FOR KEY       !wait 10 secs for key
510 WAIT FOR TIME             !force wait for remainder
520 GOSUB 5000                !now do something
```



- WAIT interrupts have four general forms as shown in the table below. Each construction of the WAIT statement is separately discussed below. Each of the four interrupt types is discussed separately elsewhere in the Reference Section.

WAIT Statement

IEEE-488

WAIT Interrupt Statements

STATEMENT FORM	MEANING
WAIT	Suspend program execution until a CTRL/C or an interrupt enabled by ON-GOTO occurs.
WAIT time expression	Suspend program execution up to the specified time limit until CTRL/C or an interrupt enabled by ON-GOTO occurs.
WAIT FOR KEY, PPOL, SRQ	Suspend program execution until CTRL/C, the specified interrupt, or an interrupt enabled by ON-GOTO occurs.
WAIT time expression FOR KEY, PPOL, SRQ	Suspend program execution, up to the specified time limit, until CTRL/C, the specified interrupt, or an interrupt enabled by ON-GOTO occurs.

WAIT
Statement

IEEE-488

Example

The following program causes the Controller to wait 10 seconds for a touch sense key entry. If no entry is sensed, execution continues. If an entry is sensed, execution continues at a subroutine. The waiting period terminates when 10 seconds have elapsed, or when the TSO is touched. The elapsed time from line 130 to the program end is reported before exiting back to BASIC.

```
100 ON KEY GOTO 500                    !do this when tso touched
110 K% = KEY                           !zero the tso
120 PRINT "touch the screen"           !tell what to do
130 T = TIME                           !get the time
140 WAIT 00:00:10 FOR KEY              !wait 10 seconds for key
150 T1 = TIME                          !get time
160 PRINT "you didn't touch anything"
170 PRINT "elapsed time = ";(T-T1)/1000;" seconds"
180 END
499 ! key touch subroutine
500 OFF KEY                            !disable the tso
510 T1 = TIME                          !get the time
520 PRINT "you touched the screen at ";KEY;
530 GOTO 170
```

When this program is run, the elapsed time varies, depending on when the TSO was touched, if at all.

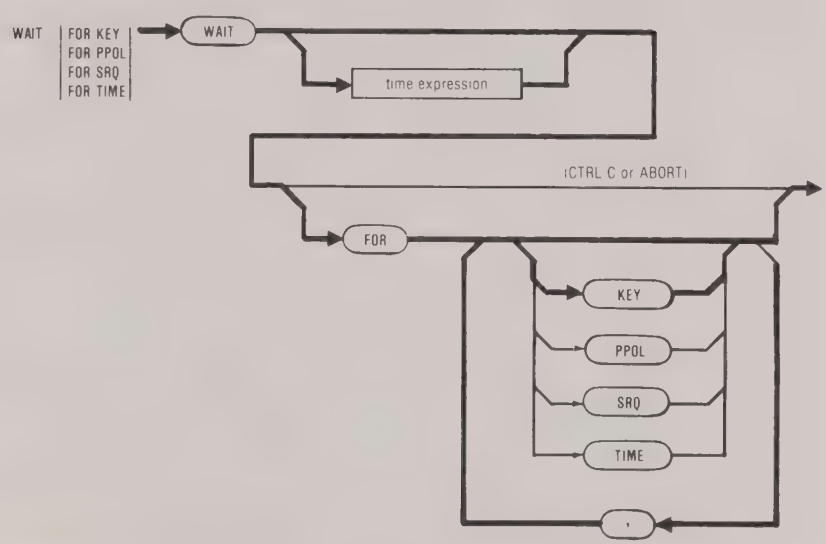
The following program is a variation on the preceeding program. This illustrates the use of the WAIT FOR TIME statement to force waiting for the full time interval specified in the previous WAIT [time expression] [FOR event] statement. When the program is run, note that the elapsed time reported is approximately 10 seconds, regardless of when the screen was actually touched.

```
10 !wait for key example #2
20 PRINT "touch the screen when ready"
30 K% = KEY \ T = TIME              !clear the tso, store begin time
40 WAIT 10000 FOR KEY \ K% = KEY !wait 10 secs, get key val
50 IF K%<>0 THEN 80 ELSE 60         !if no key...
60 T1 = TIME \ PRINT "you didn't touch anything, did you?"
70 GOTO 120
80 T2 = TIME                        !get time screen was touched
90 PRINT "you touched the screen in ";(T2-T1)/1000," seconds"
100 WAIT FOR TIME                   !wait remainder of time
110 T1 = TIME                       !get the time
120 PRINT "elapsed time = ";(T1-T)/1000;" seconds"
130 GOTO 20
140 END
```

Usage

WAIT [time expression] FOR KEY

Syntax Diagram



Description

The WAIT FOR KEY statement suspends operation of the program until the Touch-Sensitive Display is pressed in the active area.

- When the WAIT FOR KEY statement is encountered, the number stored in KEY is checked for non-zero.
- A zero value for KEY will cause program execution to stop until KEY becomes non-zero, or until the specified time has elapsed.
- A non-zero value for KEY will immediately terminate the wait condition, passing control to the next program statement.
- A time limit may be specified following the word WAIT as described under the WAIT Time Statement.

WAIT FOR KEY
Statement

- The statement will be ignored if the specified time is zero or negative.
- The wait time will be indefinite (until the Touch-Sensitive Display is pressed) if no time is specified.
- The WAIT is terminated whenever the Touch-Sensitive Display is pressed in the active area.
- A touch key input causes the program to continue with the next statement unless a previous ON KEY GOTO statement has been executed.
- When a previous ON KEY GOTO statement has been executed, a touch key input causes the program to transfer to the KEY processing routine.
- The system variable KEY is set whenever the Touch-Sensitive Display is pressed in an active area, regardless of whether WAIT FOR KEY is used. It remains set until it is read by a program statement. (For example, K% = KEY)
- Wait time is 0 if the system variable KEY is nonzero when WAIT FOR KEY is executed.
- The WAIT can be terminated in any of the ways defined above under the WAIT statement.

Example

The following example illustrates the KEY variable and the WAIT FOR KEY interrupt. Line 10 clears the KEY buffer. This resets any value it contained from touching the display before the program started. Line 20 halts the program until the Touch-Sensitive Display is touched. Line 30 prints the KEY number. Line 30 also clears the key buffer. If the buffer were not cleared, line 20 would detect it again, allowing line 30 to display the same key value repeatedly.

```
10   K% = KEY                        ! Clear key buffer
20   WAIT FOR KEY                    ! Enable key interrupt
30   PRINT 'KEY = '; KEY             ! Display key value
40   GOTO 20
```

Sample displayed results:

```
KEY = 41
KEY = 1
KEY = 23
KEY = 57
KEY = 18
```

The following example program displays the number of each key in double-size directly under the spot that was touched.

- Flag KF% in line 80 enables the interrupt routine to clear the "Touch the Display" message from the display.
- After displaying a prompt message, line 120 halts the program until the display is touched. Then line 130 enables the KEY interrupt.
- Since the KEY buffer has a number in it from touching the display, line 130 immediately branches to the interrupt routine, disabling the KEY interrupt.
- Since the KEY flag KF% is initially zero, line 220 clears the "Touch the Display" message.
- Line 230 sets the KF% flag to one, so that subsequent passes through the routine will not clear the display.
- Lines 240 through 290 compute the position of the spot that was touched and display the KEY number at that spot.

WAIT FOR KEY
Statement

- Line 300 reenables the KEY interrupt and branches back to the main routine. Since the ON KEY GOTO statement is still active, line 140 waits for another touch on the display.

- If a touch occurs within 10 seconds, the interrupt routine is reentered.

- If a touch does not occur within 10 seconds, line 150 disables the KEY interrupt and branches to line 80 starting the sequence over again.

```
10   ! *** Display Keys ***
20   !
30   ! Displays the KEY number of the spot touched. Clears
40   ! the display if not touched within 10 seconds.
50   !
60   K% = KEY                    !Clear KEY buffer
70   ES$ = CHR$(27) + "["        !Display escape sequence
80   KF% = 0%                    !KEY flag
90   PRINT ES$; "2J"; ES$; "1p" !Clear display, double-size
100  PRINT CPOS(4,10);           !Position cursor
110  PRINT "Touch the Display!" !Display prompt message
120  WAIT FOR KEY                !Wait until display is touched
130  ON KEY GOTO 200             !Enable KEY interrupt
140  WAIT 10000                  !Wait 10 seconds
150  OFF KEY                     !Disable KEY interrupt
160  GOTO 80                     !Loop
170  !
180  ! *** Key Interrupt Routine ***
190  ! 200  K% = KEY              !Get KEY number
210  IF KF% = 1% THEN 240        !First time through?
220  PRINT ES$; "2J"             !Clear screen
230  KF% = 1%                    !Set KEY flag
240  KI% = K% - 1%               !Compute KEY index
250  TR% = INT (KI% / 10%)       !Compute touch panel row
260  DR% = TR% + 2%              !Compute display row
270  DC% = 6% + (KI% - 10% * TR%) * 3% !Compute column
280  PRINT CPOS(DR%,DC%);        !Position cursor to spot touched
290  PRINT USING "##", K%;       !Display key on spot touched
300  RESUME 140                  !Wait for another key
```

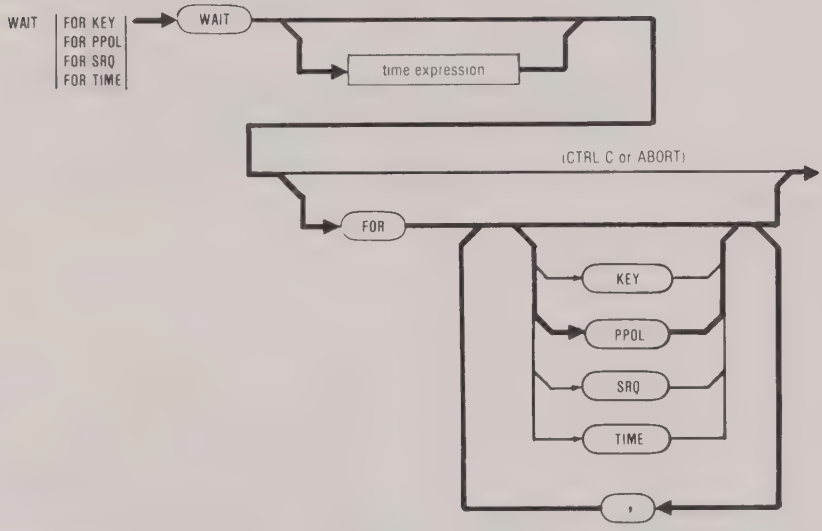
WAIT FOR PPOL 166
Statement

IEEE-488

Usage

WAIT [time expression] FOR PPOL

Syntax Diagram



Description

The WAIT FOR PPOL statement suspends operation of the program and initiates continuous parallel polling indefinitely until a positive parallel poll response is detected on the instrument port.

- The WAIT can be terminated in any of the ways defined above under the WAIT statement.
- A time limit may be specified following the word WAIT as described for the WAIT FOR [event] Statement.
- The WAIT is terminated whenever a positive parallel poll response is detected on the instrument port (both are checked if the optional IEEE-488 port is installed).
- A positive parallel poll response causes the program to continue with the next statement unless a previous ON PPOL GOTO statement has been executed.

WAIT FOR PPOL
Statement

- When a previous ON PPOL GOTO statement has been executed, a positive parallel poll response causes the program to transfer to the parallel poll processing routine.

- The program statements following WAIT FOR PPOL should cause the parallel poll response bit of the responding instrument to be reset, if possible.

- WAIT time is 0 if either instrument port has a positive parallel poll response when WAIT FOR PPOL is executed.

- See the note in the description of ON PPOL GOTO in this section.

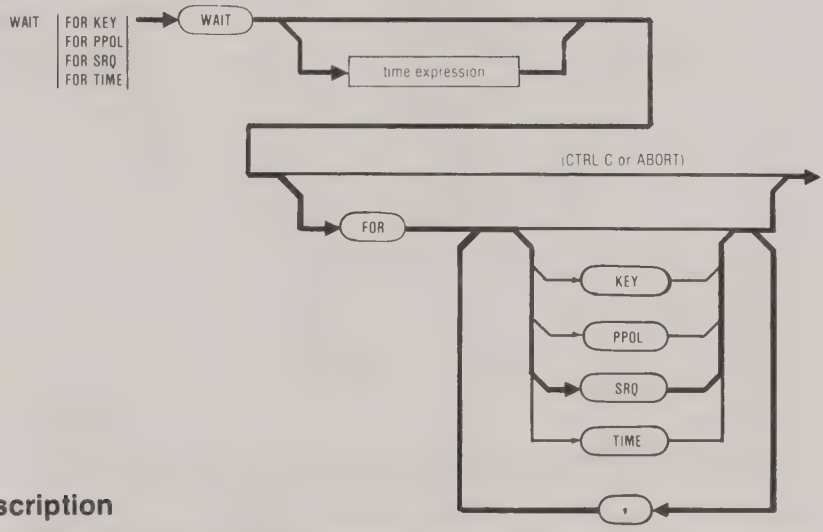
WAIT FOR SRQ Statement

IEEE-488

Usage

WAIT [time expression] FOR SRQ

Syntax Diagram



Description

The WAIT FOR SRQ statement suspends operation of the program indefinitely until a service request is detected on the instrument port. Both ports are checked if the optional IEEE-488 module is installed.

- The WAIT can be terminated in any of the ways defined under the WAIT statement.
- A time limit may be specified following the word WAIT as described for the WAIT FOR [event] Statement.
- The WAIT is terminated whenever a service request is detected on either instrument port.
- A service request causes the program to continue with the next statement unless a previous ON SRQ GOTO statement has been executed.
- When a previous ON SRQ GOTO statement has been executed, a service request causes the program to transfer to the service request processing routine.

WAIT FOR SRQ
Statement

- An internal SRQ flag is set by a service request. It is reset in the controller by performing any serial poll (for example, Y% = SPL(10)). However, depending on the instrument, SRQ will probably not be set again until a serial poll is performed on the instrument requesting service.

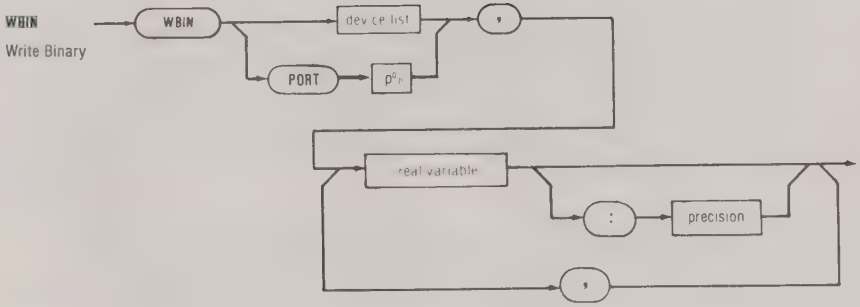
- WAIT time is 0 if the internal SRQ flag was not reset after the last service request.

- WAIT FOR SRQ does not reset the internal SRQ flag.

Usage

WBIN {device or port p%},{integer array subrange}[secondary address]

Syntax Diagram



Description

WBIN (Write BINary) sends numeric data to an IEEE-488 bus instrument in single- or double-precision IEEE standard floating point format.

- The instrument with the specified device number is addressed as a listener.
- Instrument addressing is skipped when the @ character follows WBIN without a device specified.
- Data is then transmitted in the specified format.
- Precision :4 specifies conversion to a four-byte single-precision number.
- Precision :8 specifies an eight-byte double-precision number (no conversion required).
- The default format is eight-byte double-precision.

WBIN
Statement

IEEE-488

Example

The following example addresses device 4 with secondary address 2 on port 0 as a listener, and then transmits the values of B1 and Z (elements I% through J%) in single-precision format.

```
4790    WBIN @ 4:2, B1:4, Z(I%..J%):4
```

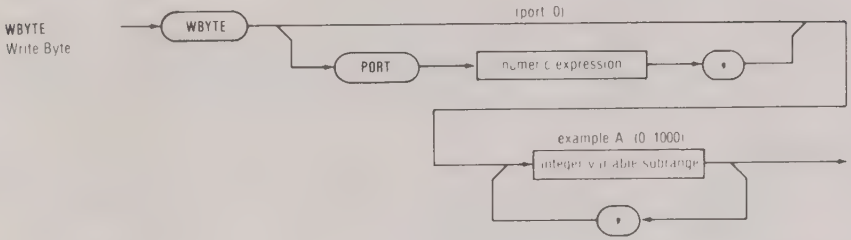
The following example addresses device 1 on port 0 and device 20 on port 1 as listeners, and then transmits the specified values from array D in double-precision format to both port 0 and port 1 (since both ports have been addressed). Values are transmitted from the array in the same order as those written in the third example for RBIN. Rows are output in column order.

```
2500    WBIN @ 1% @ 120%, D(0%..1%, 5%..7%)
```

Usage

WBYTE [PORT p%,]{integer array subrange}

Syntax Diagram



Description

WBYTE (Write BYTE) sends an arbitrary set of bus commands or data bytes to a port. The ATN, EOI, and data lines of the instrument bus may be set as desired with this command, with the restriction that ATN and EOI may not be set true simultaneously (since this causes a parallel poll).

- The array must have only a single dimension (subscript).
- The array must be integer type.
- A virtual array may not be used.
- Data transmitted by WBYTE is taken directly from specified integer arrays (one byte per array element).
- Data within each array element is formatted as follows (bit 0 is low-order):

BITS	CONTENTS
0-7	8-bit data byte
8	Send EOI if set to 1
9	Send ATN if set to 1
10-15	Ignored

WBYTE
Statement

IEEE-488

- WBYTE performs no automatic device addressing.
- The data sent from the array, however, may designate talker or listeners.
- WBYTE is used as a clause within some other IEEE-488 bus control statements. When used as a clause, it is enclosed in braces as WBYTE . . . , and may only have one subrange designated.

Example

The following example sends a binary data byte contained in array element 0 of array A% to port 1:

```
7440    WBYTE PORT 1%, A%(0%)
```

The following example sends seven data bytes from array A% (elements 1 through 7 in order) and two data bytes from array B% (elements 12 and 13) to port 0.

```
30080   WBYTE PORT 0%, A%(1%..7%), B%(12%..13%)
```

The following example illustrates one way the integer array may be defined and sent on port 0, the default port:

```
110     ! Assign array for programming instrument
120     D%(0) = 512% + 63%          ! ATN and UNL
121     D%(1) = 512% + 95%          ! ATN and UNT
122     D%(2) = 512% + 34%          ! ATN and listen address of 2
123     D%(3) = 63%                 ! "?" for trigger
124     D%(4) = 512% + 95%          ! ATN and UNL
125     D%(5) = 512% + 65%          ! ATN and talk address of 1
130     ! Send data on port 0
140     WBYTE D%(0%..5%)
```